Formal Semantics for LIPS

(**L**anguage for **I**mplementing **P**arallel/distributed **S**ystems)

A thesis submitted to Middlesex University
in partial fulfilment of the requirements for the degree of
Doctor of Philosophy

Amala Vijaya Selvi Rajan

School of Engineering and Information Sciences
Middlesex University

May 2009

Dedicated to Almighty

Abstract

This thesis presents operational semantics and an abstract machine for a point-to-point asynchronous message passing language called LIPS (Language for Implementing Parallel/distributed Systems). One of the distinctive features of LIPS is its capability to handle computation and communication independently. Taking advantage of this capability, a two steps strategy has been adopted to define the operational semantics. The two steps are as follows:

- A big-step semantics with single-step re-writes is used to relate the expressions and their evaluated results (computational part of LIPS).
- The developed big-step semantics has been extended with Structural Operational Semantics (SOS) to describe the asynchronous message passing of LIPS (communication part of LIPS).

The communication in LIPS has been implemented using Asynchronous Message Passing System (AMPS). It makes use of very simple data structures and avoids the use of buffers.

While operational semantics is used to specify the meaning of programs, abstract machines are used to provide intermediate representation of the language's implementation. LIPS Abstract Machine (LAM) is defined to execute LIPS programs. The correctness of the execution of the LIPS program/expression written using the operational semantics is verified by comparing it with its equivalent code generated using the abstract machine.

Specification of Asynchronous Communicating Systems (SACS) is a process algebra developed to specify the communication in LIPS programs. It is an asynchronous variant of Synchronous Calculus of Communicating Systems (SCCS). This research presents the SOS for SACS and looks at the bisimulation equivalence properties for SACS which can be used to verify the behaviour of a specified process.

An implementation is said to be complete when it is equivalent to its specifications. SACS has been used for the high level specification of the communication part of LIPS programs and is implemented using AMPS. This research proves that SACS and AMPS are equivalent by defining a weak bisimulation equivalence relation between the SOS of both SACS and AMPS.

Acknowledgements

I express my deep and sincere gratitude to my supervisors Dr. Geetha Abeysinghe and Dr. Siri Bavan for their outstanding guidance and encouragement throughout all the stages of this work. Both of them have been wonderful supervisors. Throughout this work, they provided encouragement, sound advice, good teaching, good company, and lots of valuable ideas. I would have been lost without their supervision and support. They taught me how to write and patiently guided me regardless of their very busy schedules. I am most grateful to both of them.

I am thankful to the School of Engineering and Information Sciences, Middlesex University, for their financial support during my study.

I am grateful to Dr. Foster, Dmitri, Dr. Ever, and Niveditha for their feedback on the draft of this thesis.

I thank my friends Dhawal, Satish, Yoney, Anjum, Lindsey, Yan, David, Yonal, and Aju. The friendship I have with them is what kept me going through tough times.

I owe my parents, Mary and Santiago, much of what I have become. I thank them for their patience, love and support. They took over a large part of my family responsibilities and encouraged me to concentrate on my studies.

I am deeply indebted to my husband Rajan for his friendship, trust, encouragement, and endurance of my bad temper.

My daughter Roshni Benedicta is my powerful source of inspiration and energy. Special gratitude is due to my nephew Rohit.

I fondly thank my sister, Jasmine and my brothers, Babu, Armstrong and Christopher and their families for their loving support.

I also thank my in-laws for their support.

List of Publications

The results presented in this thesis have also appeared in the following publications:

- A.V.S. Rajan, A. S. Bavan, and G. Abeysinghe (2008), "An Equivalence Theorem for the Specification of Asynchronous Communication Systems (SACS) and Asynchronous Message Passing System (AMPS)", accepted for the International Joint Conferences on Computer, Information, and Systems Sciences, and Engineering (CISSE 2008) to be held in December 5 - 13, 2008, to be published in Springer.

- A.V.S. Rajan, A. S. Bavan, and G. Abeysinghe (2008), "Semantics for the Specification of Asynchronous Communicating Systems (SACS)", Advances in Computer and Information Sciences and Engineering, Springer, Sobh, Tarek (Ed.), 2008, pp. 33-38, ISBN: 978-1-4020-8740-0.

- A.V.S. Rajan, A. S. Bavan, and G. Abeysinghe (2007), "Semantics for a Distributed Programming Language Using SACS and Weakest Pre-Conditions", International Journal on Information Processing (IJIP), I K International Publisher, New Delhi - 110 016, India.

- A.V.S. Rajan, A. S. Bavan, and G. Abeysinghe (2007), "Semantics for an Asynchronous Message Passing System", Advances and Innovations in Systems, Computing Sciences and Software Engineering, Springer, Elleithy, Khaled (Ed.), pp. 83-88, ISBN: 978-1-4020-6263-6.

- S. Bavan, E. Illingworth, A.V.S. Rajan, and G. Abeysinghe (2007), "Specification of Asynchronous Communicating Systems", In Proceedings of IADIS Applied Computing 2007, Nuno Guimaraes and Pedro Isaias(Ed.), pp. 274-281, ISBN: 978-972-8924-30-0.

- S. Bavan, A.V.S. Rajan, and G. Abeysinghe (2007), "Asynchronous Message Passing Architecture for a Distributed Programming Language", In Proceedings of IADIS Applied Computing 2007, Nuno Guimaraes and Pedro Isaias (Ed.), pp. 674-678, ISBN: 978-972-8924-30-0.

- A.V.S. Rajan, A. S. Bavan, and G. Abeysinghe (2006), "Semantics for a Distributed Programming Language Using SACS and Weakest Pre-Conditions", in the Proceedings of the 14th International conference on Advanced Computing and Communications, IEEE Press, pp. 434-439, ISBN: 1-4244-0715-X.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The software industry is continuously making efforts to improve the quality of distributed programming languages. Formally specifying the syntax and semantics of programming languages offer a solution towards this goal. Formal specifications present a worthwhile subject of study due to the following reasons:

- They are used in requirement specification.
- They serve as a precise standard for compiler implementation.
- They provide a vehicle for verification and validation.
- They assist in language design.
- They provide useful user documentation.

This research aims to develop formal semantics for Language for Implementing Parallel/distrubuted Systems (LIPS) [Bavan and Illingworth, 2001].

1.1 Distributed Programming Languages

Programming languages can be classified into two main groups: *sequential* and *distributed.* Sequential programming languages such as FORTRAN, Pascal, and C are executed on a single processor. Distributed programming languages such as Occam [Inmos, 1988], Ada [Ledgard, 1983], NIL [Strom and Yemini, 1983, 1985], and Concurrent C [Gehani, 1990, Gehani and Roome, 1992] consist of number of simultaneous sequential processes which can be executed on a number of processors.

Different distributed programming languages exhibit different distinct features which include parallelism, communication, fault tolerance, architecture independency, understandability, implementability, optimality, functionality, and security [Bal et al., 1989, Skillicorn and Talia, 1998, Haridi et al., 1998]. This work considers three main issues

that distinguish a distributed language from a sequential language namely: ability to handle parallelism, communication, and separation of communication and computational components. They are considered briefly below:

1. **Parallelism**: This refers to the possible methods of running more than one part of program simultaneously. One important factor to be considered while designing a programming language is what to use as the unit of parallelism. A unit of parallelism can be expressed in terms of processes, objects, statements, expressions, and AND/OR clauses [Bal, 1990].
   For example, Ada handles parallelism through sequential processes called tasks, Emerald and Smalltalk use objects, while Occam uses statements. This work views parallelism as a set of processes executing simultaneously on different processors co-operating closely by communicating with each other.

2. **Communication**: This involves interaction between processes and their synchronisation. Communication between processes can be achieved by either shared memory or message passing.

   - **Shared memory multi-processor systems**: Shared memory multi-processor systems provide a shared memory abstraction in which an application is written as if it were using a global address space. In other words, these systems are built using multiple high performance microprocessors which logically share a common memory [Stenstram and Dahlgren, 1996]. The fundamental features of shared memory are that the inter-process communication is implicit, synchronisation is explicit and the physical location of the data is completely unspecified [Kubiatowicz, 1998]. Though it is easy to program distributed applications using global address space which results in fast data sharing, shared memory systems require major communication overheads which degrade the efficiency of message passing and increase the cost. Concurrent Pascal [Brinch-Hansen, 1975], Algol 68 [Wijngaarden, 1981], Linda [Ahuja et al., 1986], Split-C [Culler et al., 1993], and Orca [Bal, 1996] are a few languages which use shared memory for inter-process communication.

   - **Message passing**: Message passing is a paradigm used to establish inter-process communications via messages explicitly [Kubiatowicz, 1998]. The processors have their own local memory. They send and receive data independently to other processors directly or through an intermediate process that mimics point-to-point transfer of data. A defining feature of the message passing model is that data (the message) transfer from the local memory of one process to the local memory of another process requires operations to be performed by both processes. Languages such as Distributed Processes (DP) [Hansen, 1978], NIL, Occam, Ada, concurrent C, Fortran M [Foster and

Chandy, 1995], PFL [Holmstrom, 1983], and Bulk Synchronous Parallel (BSP) model [Krizanc and Saarimaki, 1996] employ message passing for communication.

Both shared-memory and message passing are dominant communication paradigms. Each approach has its own advantages and disadvantages. Several studies have been carried out analysing the performance of shared memory and message passing programming [Lin and Snyder, 1990, Ngo and Snyder, 1992, Klaiber and Levy, 1994, Kubiatowicz, 1998] and researchers have come up with a hybrid distributed shared memory communication model by combining the advantages of both paradigms.

There is also another type of system based on distributed data structure. A distributed data structure is a data structure that can be manipulated by many parallel processes simultaneously [Carriero et al., 1986]. Languages such as Linda [Carriero et al., 1986] and Orca [Bal, 1996] use distributed data structures.

The work presented in this report is based on message passing and does not delve much into either shared-memory or distributed data structures. There are four main message passing models: point-to-point, rendezvous, Remote Procedure Call (RPC), and one-to-many. Point-to-point communication can be either *synchronous* or *asynchronous*. Occam passes messages in a point-to-point synchronous fashion. Ada and Concurrent C pass messages in rendezvous manner. NIL uses point-to-point message passing in either a queued synchronous or an asynchronous fashion.

In *synchronous* communications, the sender waits for the receiver to receive the message. The sender and receiver must synchronise to exchange data. In *asynchronous communication*, the sender does not wait after sending data. The communication between processes is usually buffered using buffers of unlimited size. The need for large buffers results in memory overheads and loss of data. To address this issue, Bavan et al. [2007b] have introduced a new message passing strategy, AMPS (Asynchronous Message Passing System). It makes use of very simple data structures and avoids the use of buffers. A detailed description of AMPS is given in Chapter 3.

3. **Separation of communication and computation**: Yet a further issue, when developing a distributed programming environment, is the separation of the communication and computational components. Such separation better accommodates multiple communication and computational components. Most of the languages which achieve such separation employ different techniques/tools/language constructs for each of the two parts.
   In the programming language Regis [Magee et al., 1994] the communication and computation are handled independently as below:

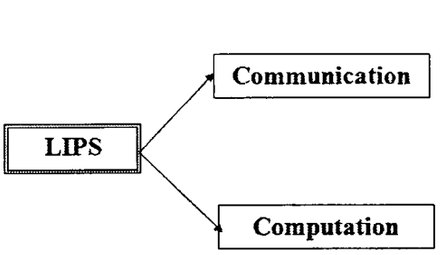
- the communication components are expressed using Darwin [Magee et al., 1993] and
- the computational elements are designed using C++.

Java has been extended with CORBA to provide a tool for developing concurrent systems [Hasselbring, 2000].

Considering the above issues, Bavan and Illingworth [2001] have taken a constructive approach to developing a distributed language to express parallelism using processes, pass messages asynchronously without message buffers, and handle communication and computational parts independently. This has led to the development of LIPS.

1.2 Language for Implementing Parallel/distributed Systems (LIPS)

Language for Implementing Parallel/distributed Systems (LIPS) is an asynchronous message passing distributed programming language which is simple and portable. One of the distinct feature of LIPS is that it handles communication and computation independently. A LIPS program consists of a network of nodes described by a network definition and node definitions. The syntactic structure of a LIPS program is shown in Figure 1.1.



Structural Syntax

PROGRAM <*identifier*>;
BEGIN
<*network_definition*>
<*node_definition*>
END.

Figure 1.1: Syntactic Structure of a LIPS Program.

The network definition describes the topology of the program by naming each node/process and its relationships (in terms of input and output data) to other nodes in the system. A node consists of one or more guarded processes which perform computations using the data that arrive as input and produces outputs that are sent to other relevant nodes.

LIPS offers distinct advantages: it is simple, portable, and it handles communication efficiently so that it avoids deadlock and livelock problems. Detailed description on LIPS can be found in [Bavan and Illingworth, 2001]. This research continues on the work already done on LIPS and seeks to develop the formal semantics and specifications which are currently lacking. This topic is expanded upon in subsequent sections.

1.3 Formal Semantics of Programming Languages

Formal semantics of a programming language is concerned with the rigorous mathematical study of meanings to programming languages and models of computation. Work on defining formal semantics for programming languages started in early 1960s [Papaspyrou, 1998, Jones, 2001]. There are number of approaches to formally specify the semantics of programming languages. They can be grouped into three categories [Andrew and Andrew, 1998] as given below:

1. **Denotational Semantics** defines the meaning of programs using suitable mathematical notations, typically functions from inputs to outputs. Denotational semantics maps a program directly to its meaning, its denotation [Schmidt, 1986].

   Denotational semantics was originally described by Scott and Strachey [1971]. It was used to devise methods for the analysis of programming languages. Further developments helped it to be used as a powerful tool for the design and implementation of programming languages [Slonneger and Kurtz, 1995].

2. **Axiomatic Semantics** defines the meaning by describing the properties about the language constructs which hold before and after the execution of the programming constructs. The properties of the language constructs are expressed in terms of predicates and deduction rules using symbolic logic and they support program verification.

   Floyd [1967] proposed a method to verify the correctness of programs by representing a program as a directed graph. Instead of specifying the programs as graphs, Hoare [1969] proposed a method of program verification which describes programs using axioms. This formed the basis for axiomatic specification. Even though Hoare's work was successful, it supported only partial correctness[1] as opposed to total correctness[2]. Dijkstra's [1976] work on weakest precondition algebra overcomes this problem as it supports total correctness.

3. **Operational Semantics** defines the meaning of programs in terms of their behaviour. For example, it describes the executional behaviour of a programming language for implementation purposes and gives a computational model for the programmers to refer to. Operational semantics, where a language is represented as an abstract machine, is used to define and implement the language [Kramer, 1994].

Denotational semantics is more abstract than operational semantics. Operational semantics gives the computational steps in the form of an algorithm whereas denotational semantics does not. Axiomatic semantics is far more abstract than denotational and

[1]Partial correctness requires that if a result is returned it will be correct

[2]Total correctness requires a result to be returned along with termination of the program

operational. Assertions and inference rules are used to define the language constructs. It is suitable for program verification. These three semantics are not in competition but they complement each other and serve different purposes. While denotational semantics and axiomatic semantics are used to reason about the programs and prove properties of programs, operational semantics is used to implement a language and prove the correctness of compiler implementation. Operational semantics is mainly used for a theoretical implementation of a language.

A relatively higher level of description of the semantics is acheived by translating the abstract syntax of a language into instructions of a simple machine using a finite collection of rules. Such a machine is called an abstract machine [Prasad and Arun-Kumar, 2002]. An abstract machine is a model of a computer system constructed to analyse how the computer system works [Hannan and Miller, 1992]. It can be proved that abstract machine is correct for its operational semantics. The correctness can be verified by checking whether the result of executing a program expression using the operational semantics matches with that of the abstract machine [Crole, 2006].

1.4 Scope of Work

This work considers the development of formal semantics for the language LIPS. In this context, we present the operational semantics and abstract machine for LIPS.

1.4.1 Operational Semantics for LIPS

Work on operational semantics started in 1960s. Landin [1963, 1965] created an abstract machine called the SECD (Stack, Environment, Code, Dump) machine to specify ISWIM (If you See What I Mean), a functional programming language [taken from [Prasad and Arun-Kumar, 2002]]. The SECD machine has been used to evaluate the Lambda expressions and formed a basis for the prototype implementations of functional programming languages [Danvy, 2003]. McCarthy's [1963, 1962, 1967] contributions during the same time period include the introduction of abstract syntax which has formed the basis for all the approaches to the semantics of programming languages.

Operational semantics was not highly regarded until radical changes were proposed by Khan, Milner, Plotkin, and others which led to a Structural approach to Operational Semantics (SOS) [Andrew and Andrew, 1998].

There are many styles of operational semantics with different terminologies and naming conventions. Natural semantics, big-step semantics, small-step semantics, transitional semantics, structural operational semantics are few example terminologies. Generally big-step semantics refers to natural semantics. But, Glesner [2003] refers to both big-step and small-step semantics as natural semantics. Peralta et al. [1998] group operational

semantics in to two categories: big-step/natural semantics and small-step or Structural Operational Semantics (SOS).

Big-step semantics describe the computations as large steps providing direct relation between initial and final states of computation [Slonneger and Kurtz, 1995] whereas, SOS describe how the individual steps of computation takes place. Big-step semantics is simple and easy to implement but it can only specify configurations related to the finite computations which makes it less suitable to specify parallelism.

SOS can

1. convey the order of execution,
2. express the properties of looping programs, and
3. reveal concurrency.

Due to these capabilities, it can be used to specify the communication part of a distributed language.

Combining the advantages of big-step and Structural Operational Semantics, a mixed two step strategy has been adopted to develop the operational semantics for LIPS. The two steps are as below:

- Firstly, big-step semantics is used to specify the expressions and their evaluated results (computational part of LIPS).
- Secondly, the developed big-step semantics is extended with Structural Operational Semantics (SOS) to describe the asynchronous message passing of LIPS (communication part of LIPS implemented using AMPS).

While operational semantics is used to specify the meaning of programs, abstract machines are used to provide intermediate representation of the language's implementation. An abstract machine is a re-writing system consisting of re-write rules to explicitly state the steps involved in the process of execution [Hutton and Wright, 2005]. It can be used to specify a way of implementing a language on some low-level computing machine or translating it to a lower-level or machine level language. The correctness of the defined abstract machine can be verified against its operational semantics. An abstract machine is considered to be correctly implemented against its operational semantics when an expression executed according to the operational semantics matches with the result of executing it with the abstract machine and vice versa.

An abstract machine called the LIPS Abstract Machine (LAM) has been defined to execute LIPS programs.The LAM was inspired by Crole [2006] and it works on the principle of re-write rules. Re-write rules are used to describe an abstract machine that maintains

a state and transforms it into a final state by repeatedly applying a given set of rules [Pingali and Ekanadham, 1988]. They explicitly show individual steps of execution and provide an intermediate level of representation for many practical implementations of programming languages [Hannan and Miller, 1992].

Using LAM the research proves the correctness of LIPS programs. This will be done by comparing the result of the code written using the LAM with the result of executing the same code written using the operational semantics.

1.4.2 Operational Semantics for the Specification of Asynchronous Communicating Systems (SACS)

Process algebra can be used to specify the communication between processes in a distributed environment. Because of its expressiveness and strong theoretical foundations, process algebra not only refers to algebraic specification but also to a method of verifying concurrent processes. Few of the well known process algebraic tools include Communicating Sequential Processes (CSP) [Hoare, 1978], Calculus of Communicating Systems (CCS) [Milner, 1982], Synchronous Calculus of Communicating Systems (SCCS) [Gray, 2000], and Language of Temporal Ordering Specifications (LOTOS) [Logrippo et al., 1990].

Since its development many extensions have been proposed for CCS to model different aspects of concurrent processing [Galpin, 1998] and Specification of Asynchronous Communicating Systems (SACS) [Bavan and Illingworth, 2000, Bavan et al., 2007a] is one of them. SACS is an asynchronous variant of SCCS which uses a point-to-point message passing system. It is developed to specify the communicating part of LIPS programs so that the communication and computation parts of LIPS can be handled independently. SACS uses the same syntax as that of SCCS but its semantics are different and governed by four design rules. It is developed by applying restrictions to the manner in which the SCCS is used and these restrictions are given in the form of design rules. These rules guarantee reliable message passing. The design rules are stated in Section 2.3.1 of Chapter 2.

Operational semantics is defined for CCS and other process algebras to precisely define the

1. set of rules for each operator in CCS,

2. execution steps that processes may engage in [Cleaveland and Smolka, 1990].

The semantics may be used to characterise the behaviour of the process algebraic description. Also, operational semantics can be used as the basis of bisimulation equivalence. Milner has introduced the concept of bisimilarities which have influenced the development of process calculi [Gordon, 1998]. Two processes are said to be bisimilar if there exists a

binary relation between the two processes such that whenever two processes are related and one can do an action, the other can match the action in such a way that the resulting processes remain related. Bisimulation is based on the idea of processes mimicking each other's behaviour [Fencott, 1996]. For example, Cleaveland and Smolka [1990] have defined an Structural Operational Semantics for CCS and shown how the defined semantics characterises the behaviour of CCS. Similarly, Fencott [1996] has defined operational semantics for CCS and Timed Calculus of Communicating Systems (TCCS) [Chen et al., 1990]. The behaviour is described using a set of Labelled Transition Systems (LTS) which consist of a collection of possible system states and transitions which have been used to study the equivalences. As the operational semantics and equivalences relations are not defined for SACS, this research considers defining the operational semantics and studying various bisimulation equivalence properties applicable to SACS.

An implementation is said to be complete one only when we prove that it meets its specifications and to prove that we need to prove an equivalence relation between the specification and its implementation. SACS has been used for the high level specification of the communication part of LIPS programs and is implemented using the Asynchronous Message Passing Systems (AMPS). It is necessary to study the proof of equivalence of SACS and AMPS to prove the completeness of AMPS. The operational semantics of both SACS and AMPS are based on Structural Operational Semantics (SOS) using Labelled Transition Systems. We then have two labelled transition system semantics: one for SACS and one for AMPS. To prove that they are equivalent, it is enough if we can prove the bisimilarity of these two labelled transition systems.

So, by defining the operational semantics for LIPS and SACS, we try to address the research question,

"Can operational semantics and SACS in combination be a suitable tool to describe the formal semantics for LIPS?"

1.5 Contribution

The main contribution of this thesis is a formal description for the semantics of the LIPS programming language. The formal semantics developed has been verified for its correctness with the main focus on the communication part. This can be found in Chapter 5 where a proof of equivalence of SACS, a tool to specify the asynchronous communication, and AMPS, its implementation, has been derived using labelled transitions. An abstract machine has also been developed and it is tested for correctness with its operational semantics. This can be found in Chapter 4. Additional contributions made as a result of this research are listed below:

- Structural Operational Semantics (SOS) and study of equivalence relation properties for SACS are described in Chapter 5. This study reveals that SACS with minor changes can be used to specify any asynchronous communicating system.
- The SOS for the Asynchronous Message Passing System (AMPS) of LIPS defined as part of the operational semantics for LIPS described in Chapter 4 will make AMPS a stand alone virtual machine which can be implemented in any asynchronous communicating applications without buffers.
- A compiler has been developed using JFlex, CUP and java. It has been tested with simple applications for its capability to implement AMPS and pass messages asynchronously.

1.6 Structure of the Thesis

The thesis is structured in the following way:

- Chapter 2 reviews the literature most relevant to the subject of investigation. This includes the following areas:
  1. Few distributed programming languages which involve message passing,
  2. Operational Semantics and abstract machine which can be used to describe distributed programming languages,
  3. Specification of Asynchronous Communicating Systems and its formal semantics.
- Chapter 3 introduces the fundamental concepts of LIPS, the language under consideration. It also gives an introduction to the Asynchronous Message Passing System (AMPS) proposed for LIPS. The AMPS of LIPS has been developed to achieve asynchronous message passing across platforms without any message buffers.
- Chapter 4 describes the operational semantics of LIPS and its abstract machine, LAM. The chapter demonstrates the correctness of the LAM with respect to the defined operational semantics.
- Chapter 5 gives a brief introduction to SACS, defines the Structural Operational Semantics (SOS) for SACS and discusses the equivalence relation properties for SACS. This chapter also verifies the asynchronous message passing implemented using AMPS against SACS.
- Chapter 6 summarises the findings and contributions of this work and discusses directions for future research.

Chapter 2

Literature Review

Designing a distributed language which can pass messages asynchronously, and handle communication and computation independently has always been a challenge and formal methods of specification are generally used for this purpose. This research is concerned with the development of operational semantics for LIPS, a Language for Implementing Parallel/distributed Systems and SACS, the Specification for Asynchronous Communicating Systems. SACS is a process algebraic framework used to specify the asynchronous communicating processes in a LIPS program.

In this chapter we review the literature relevant to the subject and is divided into the following subsections:

- Section 2.1 gives an overview on some of the parallel/distributed programming languages which use message passing for communication and justify the need for a distributed programming language like LIPS.
- Section 2.2 discusses the existing operational semantics that have been used to specify parallel/distributed programming languages and analyses the necessity for a mixed approach to specify the semantics of a distributed programming language.
- Section 2.3: discusses about the Specification of Asynchronous Communicating Systems (SACS). It also considers the correctness of specification using SACS with its implementation. The objective is to prove that the implementation created for a system involving asynchronous communication in a LIPS program meets its requirement specification created using SACS.
- Section 2.4: This section concludes the literature review.

2.1 Parallel/Distributed Programming Languages

A number of parallel/distributed languages have been proposed that employ message passing for communication. Bal's [1990] survey on programming languages for distributed

computing has discussed three main issues that distinguish parallel/distributed languages from sequential languages, namely *parallelism, communication*, and *partial failures.* Work on languages for parallel computation by Skillicorn and Talia [1998] has listed six properties that a useful parallel programming language should have, which are, *programmability, efficient methodologies, architectural independence, understandability, implementability* and *optimality.* Haridi et al.'s [1998] survey on programming languages for distributed applications is concerned with five issues while designing a distributed programming language, namely, *functionality, distribution structure, open computing*[1], *fault tolerance* and *security.* Haridi et al. also have proposed a design for a distributed programming language called Distributed Oz which separates the application functionality from its distribution structure. A review on the issues listed by these authors would be beyond the scope of this thesis. This work therefore will consider three main issues namely: ability to *handle parallelism, communication,* and *separation of communication from computation.*

This section describes these issues and reviews some of the popular parallel/distributed programming languages.

1. **Parallelism**:

   As stated in Chapter 1 parallelism refers to the possible methods of running more than one part of a program simultaneously. A unit of parallelism can be expressed in terms of processes, objects, statements, expressions, AND/OR clauses [Bal, 1990] and they are described below:

   (a) **Processes**: A process is a logic unit consisting of a set of instructions to be executed sequentially and has its state and own data. Parallelism is based on processes in many procedural languages for distributed programming [Bal et al., 1989].

   Ada [Ledgard, 1983], concurrent C [Gehani, 1990, Gehani and Roome, 1992], Linda [Carriero et al., 1986, Ahuja et al., 1986], Erlang [Armstrong, 2007], and NIL [Strom and Yemini, 1983, 1985] are some of the languages which use process as a unit of parallelism. Using the notion of process gives greater flexibility to the programmer as they can preset the communication channels between processes. But individual mechanisms are needed to set up the communication channels for passing messages and extra efforts are needed to prevent processes communicating with terminated processes. Some of the techniques used are setting the status of the channels and guarding the processes, which can be used independently or in combination.

[1]Open computing is a general and inclusive term that is used to describe a philosophy of building IT systems. In hardware, open computing manifests itself in the standardization of plug and card interfaces; and in software, through communication and programming interfaces. Open computing allows for considerable flexibility in modular integration of function and vendor independence [Heintzman, 2003].

(b) **Objects**: They are self contained units with associated data and method. Languages that use objects to structure their programs are called as object oriented languages. These languages express parallelism in two ways. One way is to use an object to express a unit of parallelism and the another way is to use the tradition notion of processes to express parallelism.

Emerald [Hutchinson, 1987] is an object-based language that allows objects fixed on a specific processor to be unfixed and moved to a different processor at runtime. Smalltalk [Horwat, 1988], an object oriented programming language, allows both objects and processes to express parallelism. Handling objects as parallel units is similar to using processes as parallel units.

(c) **Statements**: Statements can be grouped so that they can be used express a unit of parallelism. There are programming languages which allow statements to be executed either sequentially or in parallel.

Occam [Inmos, 1988] uses the keywords SEQ (sequential) and PAR (parallel) before a group of statements to express sequential and parallel executions respectively.

For example, the following code will execute statements $S1$ and $S2$ sequentially:

```
SEQ
    S1
    S2
```

The following code will execute statements $S1$ and $S2$ parallely:

```
PAR
    S1
    S2
```

It is an easy to use but an uncommon method of achieving parallelism.

(d) **Functions**: A function is a program unit which returns a single value whenever called by the main program. Functions are used in both procedural and functional languages. In functional languages like Haskell [Jones, 2003], the result of a function depends only on its input values. But in procedural languages, the result of one function may depend on one or more other functions. All function calls in a program can be executed in parallel with the exception to functions which use the result of other functions. It is not a popular method as it is not good practice to evaluate all the functions in parallel. If a parallel function is doing a simple task, the overheads involved in parallel execution may outweigh the savings in computer time.

Data flow languages such as VAL (Value-oriented Algorithmic Language) [Acherman et al., 1979] are based on this principle [Bal et al., 1989].

(e) **AND/OR clauses**: There are two methods of implementing parallelism in logic programming namely, OR and AND parallelism. OR parallelism is used when several alternative clauses for a goal are executed in parallel. AND parallelism is used when two or more goals of the same clause are executed simultaneously Ertel [1991]. This method of achieving parallelism is used in parallel logic languages such as Concurrent Prolog [Shapiro, 1986]. Apart from AND/OR parallelism, processes are also used to implement parallelism in logic programming.

Considering the above methods of achieving parallelism, it can be inferred that using processes to express parallelism is the most commonly used method which is also used by object oriented and logic programming languages.

Assigning the processes to processors can be fixed at *compile time, runtime* or *anytime* [Bal, 1990]. The advantage of assigning processes at compile time is that the developer knows which process will be running on a specific processor. But it suffers from a limitation that this method of mapping is less flexible and restricted. StarMod is a concurrent language which uses this concept [Cook, 1980]. Assigning processes for parallel processing during runtime may seem to be a flexible method but it needs extra programming to allocate and reallocate the processes to processors automatically. Concurrent PROLOG achieves parallelism using Logo-like[2] turtle programs developed by Shapiro Shapiro [1986] where each processor can communicate with four neighbour processors. Assigning processes to processors anytime allows high flexibility as one can switch between compile time and runtime methods. For example, the language Emerald [Hutchinson, 1987] uses this concept of non-mapping. Emerald is an object-based language that allows objects fixed on a specific processor to be relocated to different processors at runtime.

ii. **Communication**:

In order for the parallel/distributed programming languages to execute the processes, they must communicate and synchronise. The inter-process communication in a parallel/distributed language may take place using *shared memory* or by *message passing* which are two opposing communication models.

(a) **Shared memory multi-processor systems**:

These systems provide a shared memory abstraction in which an application

[2]Logo, a dialect of the Lisp language, is a programming language created in 1967. It was used to control a simple robot called turtle which is represented as a screen turtle on the computer screen in the recent versions [Friendly, 1988]. Each turtle has state with a position on the screen and a heading showing the direction it is facing. There are methods for moving the turtle in steps in just four directions around a grid, and for moving the turtle in all directions with pixel or better accuracy.

is written as if it were using a global address space. In other words, these systems are built using multiple high performance microprocessors which logically share a common memory [Stenstram and Dahlgren, 1996]. Languages implemented using shared memory multi-processor systems include Concurrent Pascal (Brinch-Hansen, 1975), Linda [Ahuja et al., 1986, Carriero and Gelernter, 1989], Algol 68 [Wijngaarden, 1981], Split-C[Culler et al., 1993], Orca [Bal, 1996], and Mesa [Geschke et al., 1977, Andrews and Schneider, 1983, Bal et al., 1989]. The fundamental features of shared memory are that the inter-process communication is implicit, synchronisation is explicit and the physical location of the data is completely unspecified [Kubiatowicz, 1998]. Though it is easy to program distributed applications using a global address space which results in fast data sharing, shared memory systems require major communication overheads which degrade the efficiency of communication and increase the cost.

(b) **Message Passing**:

Message passing is a paradigm used to establish inter-process communications via messages explicitly [Kubiatowicz, 1998]. The processors have their own local memory and they send and receive data independently to other processors directly or through an intermediate process that mimics point-to-point transfer of data. Languages such as Distributed Processes (DP) [Hansen, 1978], NIL [Strom and Yemini, 1983, 1985] , Occam [Inmos, 1988], Ada [Elsom, 1989], concurrent C [Gehani, 1990], Fortran M [Foster and Chandy, 1995], PFL [Holmstrom, 1983], and Bulk Synchronous Parallel (BSP) model [Krizanc and Saarimaki, 1996] employ message passing for communication. The basic component of message passing is the point-to-point communications to send and receive data between two processes. Typical point-to-point communication can be either *synchronous* or *asynchronous*.

In *synchronous* communications, the sender waits until the complete message can be accepted by the receiving process and the receiver waits until the expected message arrives. This type of message passing is also referred as blocking. Synchronous message passing does not require buffer storage. Communicating Sequential Processes (CSP) [Hoare, 1978], which is the basic message passing paradigm is an example of synchronous message passing.

Asynchronous or non-blocking message passing refers to the type of communication in which data can be transmitted intermittently. The communication between processes is buffered using buffers of unlimited size. The sender does not wait after sending the data. The receiver waits only when the buffer is empty. Most of the latest distributed programming languages have adopted

asynchronous communication. The major disadvantage of these asynchronous message passing technique is the necessity for large buffers. These message buffers must be protected, notified or interrupted when message passing is complete. Another major problem of buffering is the memory overhead. It would be ideal to develop a reliable asynchronous message passing system which does not depend on message buffers.

iii. **Separation of communication and computation**:
Another issue to be considered while developing a distributed programming environment is the separation of the communication and computational components within the program structure. Such separation better accommodates multiple communication and computation components and primitives. Most of the languages which achieve such separation employ different techniques/tools/language constructs for each of the two parts. For example, Regis [Magee et al., 1994] is a programming environment aimed at supporting the development and execution of distributed programs. The computational elements of a Regis program are designed using C++ and the communication is expressed using Darwin [Magee et al., 1993]. Distributed Oz has been designed to handle the application functionality and distribution structures separately. Java is extended with CORBA to provide a tool for developing concurrent systems [Hasselbring, 2000]. However, more research is required to improve the efficiency of such separation.

There are a number of parallel/distributed programming languages that have been developed. These languages are generally grouped based on their ability to express parallelism, pass messages, resource sharing, reliability, performance, and simple design [Tel, 2000]. Following sections review some of the popular distributed programming languages which demonstrate various ways of message passing.

2.1.1 Occam

Occam, [Inmos, 1988], is a simple concurrent low level programming language developed for transputers. Although Occam has been developed for transputers, it has also been implemented on other platforms such as VAX VMS, IBM PC compatibles and SUN workstations [Hyde, 1995, Tanenbaum et al., 1989]. The implementation is achieved by installing an additional board containing one or more transputers. Occam was derived from Communicating Sequential Processes (CSP) [Hoare, 1978] which allows the behaviour of the language to be specified more formally. CSP aims at having both guarded inputs and outputs for communication. Implementing communications that are guarded at both ends poses serious design difficulties which is one of the major setbacks of CSP. Therefore, Occam provides only guarded inputs.

The execution of processes can be either parallel (PAR) or sequential (SEQ) and must

be explicitly stated in Occam. Unlike other mechanisms which express parallelism, PAR is not a common method even though it is a natural and easy construct to use.

Communication in Occam is achieved indirectly through channels. A channel in Occam has a unidirectional link between two processes which is only available to one process at a time. The channels are typed and their names can be passed as parameters to procedure calls. The message passing is via point-to-point communication and is synchronous. It is well known that synchronous message passing causes delay in communication which may in turn affect the overall performance of the system. Though Occam is constructed as a synchronous message passing language, research to support asynchronous communication has been carried out [Serbedzija, 1988, Theodoropoulos et al., 1997, Illingworth et al., 1995].

Due to its static nature of modelling, the mapping between the processors and channels in Occam are fixed at compile time. Despite the fact that programmers take advantage of this mapping by knowing about the availability of shared memory for the various processes [Bal et al., 1989], it enforces severe restrictions on the communication and affects the flexibility of its programs [Demaine, 1996] and this is due to the fact that Occam does not permit new processors to be created dynamically during run time [Carriero and Gelernter, 1989].

To summarise, Occam is a simple language for embedded systems which,

- Communicates concurrently using channels;
- Mapping between processors and computations are fixed at compile time;
- Passes message between processes synchronously.

Occam has been used extensively for programming applications in the area of signal processing, image processing, simulation, numerical analysis and neural computing and efforts are being made to improve its performance in terms of its message passing features and static nature [Bal et al., 1989, Bal, 1996, Theodoropoulos et al., 1997, Welch and Barnes, 2005]. The explicit nature of expressing parallelism using the PAR statement increases the responsibility of the developers during the designing phase. Security in concurrent systems is an important aspect and it is supported in Occam by not having some of the widely used functions of other programming languages such as pointers, dynamic memory allocation, dynamic process allocation and recursive functions of programming languages.

2.1.2 Ada

Ada, [Ledgard, 1983], is a language designed to be used by the US Government for use in embedded systems. It is loosely based on Pascal with similar syntax and strong-typing.

Similar to Occam, Ada also is formally based on CSP [Fidge, 1993] but not too closely due to its rendezvous nature of communication. The rendezvous model of communication[3] is based on three constructs which are the entry declaration, the entry call and the accept statement.

Parallelism is achieved through sequential processes known as tasks. The tasks in Ada are typed. Each task is defined using two parts:

- Task specification - specifies the name of the task and formal parameters that define the communications interface of tasks of that type
- Task body - defines its execution

Tasks can be created dynamically. Ada was intended to generate embedded systems and there is no notation to map the tasks to processors, [Jansohn, 1988, Bal et al., 1989]. The main disadvantage of this type of mapping is that it will be expensive to identify the operations that may need inter-process communication. Therefore, Jansohn [1988] proposed to write several Ada programs, one for each processor and implement the required communication software.

Communication between the tasks is defined and synchronized by the rendezvous model. A task may call (entry call) another task by specifying the entry point. The caller task synchronises with the called task using the accept statement. This is similar to the Remote Procedure Call (RPC) where the entry point and the accept statement are on the server side and the entry call is on the client side, where the entry call is similar to a procedure call. Java and concurrent C support this type of rendezvous communication. When the entry call synchronises with the entry point, the two tasks merge to execute a guarded code [Carlson et al., 1980]. Tasks can be terminated if the rendezvous does not occur. Several tasks may call an entry before a corresponding 'accept' statement is reached and in that case the calls are queued. Each execution of an accept statement will remove a call from the queue in a First In First Out order (FIFO) of arrival [Brauer et al., 1981].

Failure detection is possible with Ada's exception handling mechanism. The standard library of Ada supports portability and it gives the flexibility to the user to add user defined libraries into the language.

Burns et al. [1987] referenced in [Bal et al., 1989] has reviewed the problems of parallel and distributed programming in Ada. One of the issues criticised by Burns is the synchronisation mechanism as it is asymmetric, entry calls are served in FIFO order and

[3]An interaction between two processes S and R is called rendezvous when S calls an entry of R, and R executes an accept statement for that entry. The interaction is fully synchronous so that the first process that is ready to interact waits for the other. When the two processes are synchronised, R executes the do part of the accept statement [Bal, 1990]

cannot be accepted conditionally.

According to Bal et al. [1989] and Andrews [1982], implementing some of the aspects of concurrent programming of Ada is complex. Rising [1988] has identified that the complexity in developing concurrent applications stem from the interaction between the tasks. Burns et al. [2001] acknowledged the problem of tasking in Ada and proposed design abstractions such as atomic actions, conversations etc to handle it. The logical correctness of these abstractions has been validated using Petri nets.

2.1.3 Concurrent C

Concurrent C, [Gehani, 1990, Gehani and Roome, 1992], is an extension of C to implement distributed processing based on rendezvous message passing. This is one of the first concurrent languages based on C which has been influenced by the concurrent facilities of Ada. The C compiler is added with run-time and system libraries which are used to translate the Concurrent C programs to C programs. The local C compiler then compiles the converted programs.

Parallelism in Concurrent C is achieved through sequential processes similar to tasks in Ada. Each process consists of a specification part and a body. The specification part consists of the name of the process, a list of formal parameters and a list of transactions. Processes are created explicitly and a newly created process can be given a priority and assigned to a specific processor.

A program in Concurrent C comprises one or more processes working together to reach a solution. Two processes interact by synchronising with each other and then exchanging information between them and continuing with their individual activities. This synchronisation to exchange information is called rendezvous communication and it is similar to Ada. A transaction in Concurrent C is different from Ada by its capability to return values. That is, Concurrent C permits two way information transfer during rendezvous communication which is called extended rendezvous. In addition, it supports asynchronous message passing. Concurrent C allows its processes to use shared memory without portability.

Concurrent C can [Bal et al., 1989, Gehani, 1990, Gehani and Roome, 1992]

- Define and create processes - the creative primitive is used to create processes explicitly and pass the parameter to the created process. The process which gets created can be prioritised and can be assigned to a specific processor;

- Specify querying and changing process priorities and accepting transactions in a user-defined order - accepting transactions conditionally based on the values of their parameters which is not the case in Ada as it has no conditional accept statement and it strictly follows FIFO queuing;

- Specify timed bidirectional synchronous transactions similar to Ada's timed entry call and ordinary/non-timed unidirectional asynchronous transactions with no return value;

- Handle interrupts and terminate processes collectively - Ada uses declarations to associate interrupts with transaction calls whereas Concurrent C uses library functions to make this association. This feature makes it possible to change or discontinue the association at any time which is not the case in Ada;

Similar to Ada, parallelism, decomposition of programs into distributed processes, mapping of processes with computations, and communication and synchronisation have to be specified explicitly. This makes the process of developing programs difficult as correctness and performance of programs must be achieved by considering a number of factors [Skillicorn and Talia, 1998].

In summary, Concurrent C is like Ada in expressing parallelism using processes and it supports both rendezvous type of message passing and asynchronous message passing. Designers of the language tried to avoid the complexity of Ada and maintain the concurrent features but Concurrent C does not support shared memory communication and therefore, it does not have semaphores, condition variables and monitors [Kamran, 1996]. It has been implemented on a UNIX operating system where it is considered as a sequential program. Context switching and scheduling have to be provided separately apart from the UNIX scheduler. In order to add object oriented programming capabilities to its concurrent programming capabilities, Concurrent C has been merged with C++ [Gehani and Roome, 1992]. But Kamran [1996] has found that integrating data abstraction facilities of C++ and concurrency features of Concurrent C was not a successful experience.

2.1.4 NIL

NIL [Strom and Yemini, 1983, 1985] is a general purpose high level programming language developed at IBM to support the construction of distributed programs. Programs developed using NIL have no pointers or data models. The inter-process communication can be either synchronous (rendezvous or RPC) or asynchronous [Strom and Yemini, 1983].

Parallelism is achieved through inter-process communication. A NIL program consists of dynamically evolving network of loosely coupled processes which encapsulate the data and its state. NIL supports point-to-point communication through channels. The communication channels in NIL are unidirectional and are created dynamically by connecting input ports with output ports. The synchronous and asynchronous communications are different from the concepts of CSP and CCS as all communications are queued and there is no sharing of data across the communication channels [Strom and Yemini, 1983]. At

any time a data object can only belong to exactly one process and a data object can be passed from one process to another. Processes contain only local data which also includes the input/output ports. Communication ports can be created by the interconnection of these ports and NIL processes can interact with each other only by these communication ports. For a NIL program, the environment is determined at run-time and therefore the choice of which modules to load into a component is also made during run-time.

NIL supports exception handling to manage software failures and it helps the developer in detecting the errors automatically. This is not the case in Occam where there is very little information available through static examination of the program. The process model of NIL is similar to Ada and Occam. The mapping of processors to processes is dynamic in NIL whereas it is fixed and static in the case of Ada and Occam [Goldszmidt et al., 1988].

To summarise, NIL is a general purpose distributed programming language which supports inter-process communication through point-to-point message passing. NIL supports only queued synchronous and asynchronous communication. Despite all these useful features, NIL needs substantial amount of run-time support and there is no evidence of a broader range of applications developed using NIL.

This section briefly looked into some of the distributed/parallel programming languages that have influenced the way in which distributed and parallel systems can be programmed. Although the languages listed here have their own short comings, each of them offer a number of features, which allow programmers to develop distributed applications. These features include:

- Point-to-point communication through inter-process communication;
- Dynamic mapping of parallel processes to physical processors;
- Simple and reliable asynchronous message passing without using buffers.

Considering all the above issues, a constructive approach to developing a distributed language is to incorporate all of the above properties into a single programming language together with additional properties like simplicity, expressiveness and support of portability.

LIPS (Language for Implementing Parallel/distributed Systems) has been developed to address these issues but currently fails to address the problem of dynamic mapping of processes to processors. However, LIPS offers many distinct advantages which includes asynchronous message passing capability, avoidance of deadlock and livelock, ability to handle communication and computation independently, and portability. The Asynchronous Message Passing System (AMPS) [Bavan et al., 2007b] implemented in LIPS allows it to pass messages asynchronously without buffers. A detailed description of LIPS can be found

in [Bavan and Illingworth, 2001]. This research continues on the work done already on LIPS and seeks to develop the formal semantics and specifications which are currently lacking. The subsequent sections look at the methods to define the formal semantics and high level specification of a distributed language such as LIPS.

2.2 Operational Semantics and Abstract Machine

Having identified the need for a distributed programming language like LIPS, this section gives the background information on operational semantics and analyses the necessity for a mixed approach to specify the semantics of a distributed programming language.

Operational semantics of a programming language gives a mathematically precise definition of how to execute programs. For example, it describes the executional behaviour of a programming language for implementation purposes and gives a computational model for the programmers to refer to.

Operational semantics is used to implement a language and prove the correctness of compiler implementation. An interpreter, which is the end product of operational semantics, is used to define the meaning of the program. It is abstract but it has the major advantage that once the interpreter has been developed, the language can be easily implemented. Operational semantics is mainly used for the following purposes [Andrew and Andrew, 1998]:

- **Prove properties of programs**: Operational semantics is used to define notions of semantic equivalence of programs and to develop the theory of such notions. A few examples include Gordon [1998]'s work on the operational equivalence of untyped and first-order languages to deal with polymorphic object calculi, and Jeffrey [1998]'s work on the definition of operational semantics and higher-order bisimulation for a subset of Concurrent ML (CML).

- **Verify the correctness of interpreters and compilers**: This is done to check the extent to which the meaning of programs is preserved and reflected against the requirement specification. An example is Jeffrey's [1995, 1998] work where subset of CML is translated to Concurrent Monadic ML (CMML). His work is correct only up to weak bisimulation.

- **Study the efficiency in terms of using temporary storage**: The operational semantics and the abstract machine are used to establish the soundness of memory management techniques. Morrisett and Harper [1998]'s work on semantics of memory management for polymorphic languages evaluate the polymorphic functional program using an abstract machine.

The description of how a program gets executed is given in a sequence of computational steps using transition systems. A transition system consists of a set of states and transitions between the states. It performs the computations allowing transitions between states. Transition system can be labelled or unlabelled. Transition systems can be represented using rules to define a transition relation (operational semantics) or as an abstract machine where the transitions represent the changes of state in the abstract machine. Operational semantics describe the steps that an abstract machine performs when running a program. An abstract machine consists of a state and an evaluation relation which forms the transition system [Fernandez, 2004]. The state of the abstract machine consists of a stack, memory and the code to be evaluated. It is the responsibility of the transition function to map from one state to another by executing the code.

The correctness of the defined abstract machine is usually verified against its operational semantics. An abstract machine is considered to be correctly implemented against its operational semantics when an expression executed according to the operational semantics matches with the result of executing it with the abstract machine and vice versa [Crole, 2006].

Work on operational semantics started in 1960s. Landin [1963, 1965] created an abstract machine called the SECD (Stack, Environment, Code, Dump) machine to specify ISWIM (If you See What I Mean), a functional programming language [taken from [Prasad and Arun-Kumar, 2002]]. The SECD machine has been used to evaluate the Lambda expressions and formed a basis for the prototype implementations of functional programming languages [Danvy, 2003]. McCarthy's contributions during the same time period include the introduction of abstract syntax [McCarthy, 1962] which has formed the basis for all the approaches to the semantics of programming languages. McCarthy's other contributions include basis for a mathematical theory of computation [McCarthy, 1962] and proof of the correctness of a simple compiling algorithm for compiling arithmetic expressions into machine language [McCarthy and Painter, 1967]. Operational semantics was not highly regarded until radical changes were proposed by Khan, Milner, Plotkin, and others which led to a Structural approach to Operational Semantics (SOS) [Andrew and Andrew, 1998]. A brief bibliography on operational semantics can be found Plotkin [2003].

There are many styles of operational semantics with different terminologies and naming conventions. A few examples are: Natural semantics, big-step semantics, small-step semantics, transitional semantics, and structural operational semantics. Generally big-step semantics refers to natural semantics but, Glesner [2003] refers both big-step semantics and small-step semantics as natural semantics. Peralta et al. [1998], groups operational semantics into two categories: big-step/natural semantics and small-step semantics or Structural Operational Semantics (SOS).

2.2.1 Big-step semantics

Big-step semantics is also known as evaluation semantics and it describes a computation in large steps providing a direct relation between initial and final states [Slonneger and Kurtz, 1995]. It ignores non-terminating computations as this will require an infinite number of derivations [Mosses, 2005]. Big-step semantics specifies the structure of terminating computations from initial configurations as a partial ordered set of state spaces. Let S be the program statements and E be the computational steps. The big-step semantics are modelled using a relation where the statement S transforms the initial state to final state inductively.

Evaluation of expressions E_1 and E_2 can be in any order or in parallel. We assume the meaning of multiplication is known. The formal rule or axiom for natural multiplication can be derived as follows:

$$nat - mul \ \frac{E_1 \Downarrow V_1 \ E_2 \Downarrow V_2}{E_1 * E_2 \Downarrow V_1 * V_2}$$

$E \Downarrow V$ means that E evaluates to V. $\Downarrow$ denotes the relation on the set of configurations E and V. V is the final configuration reachable from E. When specifying the concrete syntax for the language, the order of execution of arithmetic operations in an expression matters but after conversion to abstract syntax, the sub-expressions can be evaluated in any order. Thus we write $8 - 3 \Downarrow 5$ to state that the pair $(8 - 3, 5)$ is a binary relation. In the same way, we can add axioms for addition, subtraction and division.

In this example, the numbers 8, 3, 1, and 6 are known values. The pairs (8, 8), (3, 3), $(8-3, 5)$, (1, 1), (6, 6), and $(1 + 6, 7)$ are all in the binary relation. Therefore, each number needs an axiom. $8 \Downarrow 8$, $3 \Downarrow 3$, $1 \Downarrow 1$, and $6 \Downarrow 6$ are all axioms. Generalising it will yield the following axiom: n is any number. This will generate an infinite set of axioms by replacing n with any number. These axioms are expressed using inductive definitions. The pairs in the relation $\Downarrow$ should be supported by proof trees.

To summarise, big-step semantics "describes how the evaluation of expressions and statements affects the program state, and, in the case of an expression, what is the resulting value" [Strecker, 2002]. But it does not show the internal steps involved in the evaluation of the result. Detailed information about natural/big-step semantics can be found in [Pettersson, 1999].

2.2.2 Small-step semantics

The small-step operational semantics is called Structural Operational Semantics (SOS) [Plotkin, 1981]. SOS allows transitions to be labelled [Mosses, 2005]. A Labelled Transition System (LTS) consist of a set of rules which can be used for the derivation of

computational steps in a given program. SOS for a program contains not only the description about the current state of the program but also the part of the program which will be executed after the current transition. The LTS rules specify how the abstract machine moves from one state to another [Peralta et al., 1998].

The SOS descriptions are deductive logic as opposed to big-step semantics which is inductive. It models how the individual steps of the computation take place representing each intermediate stage as a well formed program phrase. For example, the following evaluation is an informal description showing the sequence of transitions:

$$(7+3) * (4+7) \rightarrow 10 * (4+7) \rightarrow 10 * 11 \rightarrow 110$$

The transitions are defined using " inference rules consisting of a conclusion that follows from a set of premises, possible under control of some condition" [Slonneger and Kurtz, 1995]. The general form of an inference rule is the premises are listed above the horizontal line and the conclusion is written below and can be denoted as $\boxed{\frac{premises}{conclusion}}$.

Sometimes, a condition under which the rule is applicable gets added to it and the inference rule written as $\boxed{\frac{premises}{conclusion}\ condition}$. If there are no premises, the horizontal line gets omitted. The inference rule which is now the axiom can be written as $\boxed{\frac{}{conclusion}}$.

Consider an example of evaluating an arithmetic expression. Let E_1 and E_2 be two arithemetic expressions to be added and s is the state of the system at any point of execution. The state transitions for adding the E_1 and E_2 are defined by the following rules:

$$\frac{(E_1,\ s) \rightarrow (E_1',\ s)}{(E_1 + E_2,\ s) \rightarrow (E_1' + E_2,\ s)} \tag{2.1}$$

$$\frac{(E_2,\ s) \rightarrow (E_2',\ s)}{(n_1 + E_2,\ s) \rightarrow (n_1 + E_2',\ s)} \tag{2.2}$$

$$\frac{}{(n_1 + n_2,\ s) \rightarrow (n,\ s)} (where\ n = n_1 + n_2) \tag{2.3}$$

Rules 2.1 and 2.2 take the form $\boxed{\frac{premises}{conclusion}}$ and rule 2.3 is an axiom with a condition.

Let S be the program statement and E be the computational states. If E1 and E2 are the initial and final states, the big-step/natural semantics will be modelled to represent the relation as big-step (S, E1, E2) showing that statement S transforms the initial state E1 to final state E2. But the small-step semantics/SOS are modelled by the relation of the form small-step (S1, E1 , S2, E2), where S1, and S2 are two statements and E1 and E2 are the output states of these two statements. This representation states that execution of statement S1 in state E1 is followed by the execution of the statement S2 in state E2.

The evaluation in SOS is driven by the syntactic structure of the programs which makes it a powerful tool to analyse the semantics using structural induction. While the big-

Table 2.1: Comparison big-step semantics and small-step semantics.

	Big-step semantics	**Small-step semantics**
Modelling local variables declarations and procedures	Easy	Execution stack is needed
Express parallelism	Succinct description for sequential programming but cannot express parallelism	Express parallelism by using interleaving steps
Order of evaluation	Does not specify order of evaluation	Explicit
Equivalence of the result of evaluation	Shows the same evaluation as that of small-step in the case of legal programs	Show same evaluation as that of big-step semantics in the case of legal programs

step semantics describe the overall results of the executions, the small-step semantics describe how the individual steps of the computations take place. In big-step semantics, the execution of the statement is described by one big transition stating its initial and final states. The derivation tree explains the reason for the transition. But in small-step, the execution of the statement is described by one or more transitions. The derivation sequence is vital and individual steps in the derivation sequence are justified by the derivation tree. When local variable declarations and procedures can be modelled easily in big-step semantics, structural operational semantics require an execution stack. The configurations for the SOS are the same as those of big-step semantics but have the following advantages as the emphasis is on individual steps of the execution of a program:

- Describing small-steps convey the order of execution of individual steps
- Describing the small-step efficiently expresses the properties of looping programs
- Small-step semantics can be used to reveal concurrency.

Big-step and small-step semantics are two different styles which can be used to specify the operational behaviour of a programming language. They have distinct features associated with them as shown in Table 2.1, yet, they express the same body of knowledge. Overall, big-step semantics is simple and easy to implement and it has been successfully used in Standard MetaLanguage (SML) [Milner et al., 1990]. A revised definition of SML [Milner et al., 1997] demonstrates that big-step semantics specifications are very stable[Glesner, 2003]. The major drawback with this type of semantics is that it can only specify configurations related by finite computations which make it less appropriate for specifying parallelism. Due to this inadequacy, it has been a trend to use a mixed approach while defining the operational semantics for languages which include parallelism and concurrency or any other special features [Attali et al., 1996, Albert et al., 2002].

- Albert et al. [2002] defined operational semantics for functional logic languages by using the big-step semantics in natural style to relate expressions and their evaluated results and extended it with small-step semantics to cover the features of modern functional logic languages;

- The semantics for SML has been generated by integrating the concurrency primitives with process algebra [Berry et al., 1992]. It has been done in two steps: the first step is to show the behaviour of sequential programs, and the second step defines the non-functional features of SML in terms of processes and integrating the primitives with these definitions. It has been proved that this approach helps to create simple semantics so that new primitives to SML can be added without disturbing the functional features and the stores can be represented as processes with the required behaviour;

- Big-step semantics has been extended with a relational approach to handle concurrent languages [Mitchell, 1994].

From these studies it is clear that operational semantics with an abstract machine can be used to define the behavioural specification of any programming language. While operational semantics is used to specify the meaning of programs, abstract machines are used to provide intermediate representation of the language's implementation. Abstract machines can be distinguished from operational semantics as they consist of simple and direct algorithmic implementations which can be used to develop compilers for the programming languages. The above mentioned study made it clear that the semantics of any programming language can be specified using more than one style of operational semantics to accommodate special features of a language, for example, concurrency in the case of a distributed programming language.

The following section reviews the formalisms suitable for the specification of the communication and computational part of any distributed programming language.

2.3 Formalism for High Level Specification of Parallel/Distributed Languages

A formal specification is a concise description of the behaviour and properties of a system written in a formal language. System specifications using formal languages are becoming vital for designing, validating, documenting, reusing, and re-engineering software systems. A formal specification language contains a set of symbols and grammatical rules to define well-formed formulae. These rules characterize the syntactic domain of the language which forms the theoretical foundation for any programming language. Formal methods

have been employed in the specification of software systems since the early days of computer science. Some of the well known formal specification methods include Z [Jacky, 1997], CSP, Vienna Development Method (VDM) [Bjorner and Jones, 1978], Larch [Guttag et al., 1993], CCS, and Formal Development Methodology (FDM) [Cheheyl et al., 1981].

Formal methods are used mainly in the following three activities [Jacky, 1997]:

- Modelling - to describe and predict program behaviour;
- Designing - to organise the internal structure of a program;
- Verification - to confirm that the code will behave as intended.

Whatever the type of activity, the method should provide the means for specifying a system so that consistency, completeness, and correctness can be assessed in a systematic manner. The aim of this section is to identify and analyse the specific formalisms and techniques that can specify the high-level operations of parallel and distributed applications correctly and clearly so that we can specify the requirements and use them for the verification of a parallel/distributed programming language.

A constructive approach for developing programs for a distributed programming environment is to separate the program structure from the communication and computational part of a program. This will easily accommodate multiple parallel and computational components and primitives. It helps in aiding the powerful concept of reusability of components. For example, as mentioned in Chapter 1, the language Regis, [Magee et al., 1994], is divided into two sections so that the communication and computational elements can be programmed individually. The computational elements are designed using C++ and then extended with the configuration language Darwin [Magee et al., 1995] to express the communication between processes. Occam has the sequential and the parallel computations distinguished with the keywords SEQ and PAR respectively. Whether or not a distributed language explicitly expresses its communication and computational components, it is possible to identify them individually. The work presented in this section considers the formalisms which are more suitable to specify the communication part of the distributed programming language, LIPS.

2.3.1 Specification of Communication Part of Distributed Languages

Concurrent systems can be described in terms of many different constructs for

- creating processes
- exchanging information between them and

- managing their use of shared resources.

This variability has given rise to a large class of formal systems called process algebra [Pie, 1995]. It is usually constructed from a set of basic processes and a set of operators. Each operator has a fixed arity, (number of arguments that a function can take), indicating the number of its operands [Best et al., 1998]. Several notations and formalisms for Process algebra have been defined. Few of the well known process algebraic tools include Milner's CCS, CSP, Synchronous Calculus of Communicating System (SCCS) [Gray, 2000] and the Language of Temporal Ordering Specifications (LOTOS) [Logrippo et al., 1990].

Process algebra offers an alternative to model checking. The term 'process algebra' does not only refer to algebraic notation for transition systems, but also to a method of verifying concurrent systems. This could be considered as an efficient method of specification at an abstract level and verification because of its expressiveness and strong theoretical foundations. In continuation with his work on CCS, Milner developed pi-calculus which introduced a new way of modelling communication that reflects its position [Milner, 1999]. For example, such ability to model process mobility is useful for describing how mobile phones communicate with different base stations when a person is on the move. Join Calculus is yet another process calculus which is a member of pi-calculus. It has also been used to model distributed and mobile programming but it avoids defining communication constructs like rendezvous communications which are difficult to implement [Fournet and Gonthier, 2000]. It has a direct embedding on ML programming language and it supports local synchronisation which means that messages always travel to set destinations and can interact only after they reach that destination. This kind of synchronisation is useful for pattern matching and function binding. Yet another asynchronous variant of pi-calculus is distributed join calculus which allows mobile agents moving between physical sites [Maludzinski and Dobrowolski, 2007]. JoCaml system is an experimental extension of the Objective-Caml language with the distributed join-calculus programming model [Fournet et al., 2002].

Signal Calculus (SC) is another process calculus specifically designed to describe coordination policies of services distributed over a network [Cardelli and Gordon, 1998, Milner, 1991]. It also is based on pi-calculus. Java Signal Core Layer (JSCL) [Ferrari et al., 2006] is a framework defined based on SC. JSCL is a middleware which formally describes coordination of distributed services based on an event notification paradigm.

Our main focus in this research is to specify the asynchronous communication part of LIPS. A process algebraic tool known as Specification of Asynchronous Communcating Systems (SACS)[Bavan and Illingworth, 2000, Bavan et al., 2007a] has been developed for this purpose. The design techniques of SACS allow the programmer to develop programs that are virtually free of livelock and deadlock conditions. An introduction to SACS is given in the following section.

SACS:

SACS, which uses point-to-point message passing, is an asynchronous variant of SCCS. It uses the same syntax as that of SCCS but its semantics are different and governed by four design rules. The main aim of SACS is to separate communication from computation so that these two activities can proceed independently. The formal notation of SACS has been derived by applying restrictions to the manner in which the SCCS is used. These restrictions are specified as four design rules. Thus, SACS has the same syntax as that of SCCS but its semantics are different. The four design rules guarantee reliable message passing and have their origins in LIPS.

The Four Design Rules

Rule 1 - Every process agent in a concurrent system operates in an iterative fashion and obeys the SACS- TEMPLATE which is as follows:

```
Process agent_1 =
  input_ch1_1[.| : |+]input_ch1_n]: Process_agent_1Bdy_1
        + ... + [input_chk_1[[.|:|+]input_chk_n]: Process_agent_1Bdy_k]
Process_agent_1Bdy_i =
         output_ch1_1 [[.|:|+]output_ch1_n]: [@|Process_agent_1]
where i ranges from 1 to k.
```

This means every node in the system should consist of one or more guards that contain input ports followed by a body of the code and then by output ports. Rule 1 implicitly states that a concurrent system is made of autonomous processes that intercommunicate using message passing.

Rule 2 - A channel must have one input port and one output port only.

Point-to-point communication is being used for this style of design. Thus fan-in and fan-out are special examples that must be specified explicitly.

Rule 3 - Within a node, an input channel can be used only by one guarded process.

If an input channel is shared by more than one guarded process, it may lead to partial or total starvation of at least one of the guarded processes. It may also lead to deadlock due to the propagation effect created by starvation. The following example:

$$A=\delta y!:A\_BDY\_1 + \delta y?:A\_BDY\_2$$

is invalid in our modified use of SCCS notation since the same channel y is being used as input guard for two different processes within a node.

Rule 4 - A self-contained system must include at least one idler operator in an input channel position.

A self-contained system is one which does not have channels which cross the application domain. The implication is that a non self-contained system will have one or more channels with just one port.

The following case compares the SCCS with SACS with its four design rules. Consider a simple case in which two processes exchange data. We can express this in SCCS notation as follows:

```
A = x! : y? : A
B = x? : y! : B
```

The above definitions state that process A outputs a value x and waits for a response on channel y from process B. There is no restriction on combining the input and output channels at the same location within a SCCS specification. The equivalent definition using the same notation in our approach is specified as shown below:

```
                    A = 1 : A_BDY_1 + y? : A_BDY_2
//1 before the ':' denotes an idle event
                    A_BDY_1 = x! : A
                    A_BDY_2 = 1 : A
                    B = x? : B_BDY_1
                    B_BDY_1 = y! : B
```

This means both processes, A and B, deliver their output on virtual channels, x and y, and call themselves. The input channels guard is a computational unit of code (a body of code). When each input channel in a specific guard has received an item of data, the associated body of code is executed. In the example given above B_BDY_1 is a body of code guarded by x?. Each body of code transforms the input data into output data and is passed onto other nodes in the system. Input channels occur before a body of code, which occurs before output channels. SACS has been designed for LIPS but also can be used for other distributed programming languages with minor amendments.

2.3.2 Formal Semantics for SACS

SACS, one of the process calculi, has been considered as a formal framework for specifying distributed systems and their behaviour. Klin [2004] considers three key aspects when formally describing systems and processes. These aspects are syntax, behaviour and process equivalence where syntax refers to structure of processes, behaviour refers to the kind of actions the processes may take and process equivalence describes those processes whose behaviours should be considered the same. These three aspects of SACS can be presented via its Structural Operational Semantics (SOS). SOS defined using the Labelled Transition system (LTS) is considered to be the standard way to give formal semantics of

concurrent programs and systems. Based on the LTS, many different equivalence relations can be defined. Equivalence relations provide a powerful tool to verify the behaviour of the defined processes. One of the finer equivalences is the bisimulation equivalence, [Tuosto, 2003], and others include trace equivalence, testing equivalence etc [Klin, 2004].

2.3.3 Verifying the correctness of SACS specification

Semantics of formal specification languages provide the foundation for their verification methods. Many researchers have contributed to verifying the correctness of two specifications. For example:

- Calkin et al. [1994] has defined a proof of equivalence of the operational and temporal semantics for real-time, concurrent programming language. It is done by defining the labelled transition system semantics through the timed transition systems and SOS and proving the bisimilarity of these two labelled transition systems.
- Cardell-Oliver [1998] has derived an equivalence theorem for the operational and temporal semantics of real-time concurrent programs.
- von Oheimb [2000] has proved that axiomatic semantics defined for Java_light is sound and complete with respect to its operational semantics.
- Two formal semantic descriptions of first order functional logic programming have been compared and equivalence relations have been proposed by López-Fraguas et al. [2007].
- In his book, Winskel [1993] has provided a denotational semantics for an imperative language called IMP and created a proof of its equivalence with operational semantics.

Similarly, it is useful to verify the correctness of SACS specification with its implementation. This can be done by creating an equivalence relation between their labelled transition system. We consider the weak bisimulation equivalence between SACS and AMPS. The reason is that when we compile a language or specification to another, it is very unlikely that we can faithfully preserve the operational semantics. This means that a transition from $P' \xrightarrow{\alpha} Q'$ in SACS may become a sequence of transitions in AMPS, namely $P' \xrightarrow{\alpha 1} \ldots \xrightarrow{\alpha n} Q'$ where most of $\alpha 1, \ldots, \alpha n$ are silent transitions. It might also be that we may not reach Q but a process equivalent to Q. In such situations, the weak bisimulation which compares only the external behaviours of the processes is preferred.

2.4 Summary

This chapter looked at some of the popular parallel/distributed languages. Each of them, which has been developed for a specific purpose, offer a number of features in terms of expressing parallelism, passing messages, mapping of processes to processors, and structural separation of the communication and computational components. It is appropriate to have a distributed language which can use processes to express parallelism, pass messages asynchronously, dynamically map the processes to processors and handle communication and computational components independently. LIPS, the Language for Implementing Parallel/distributed Systems, is a language which addresses most of these characteristics.

The software industry is continuously making efforts to develop high quality programming languages and formally defining the syntax and semantics offer a solution towards that goal. In pursuit of this aim, this work focusses on developing a formal semantics and specification for LIPS.

As far as the formal semantics for LIPS is concerned, the study looks into ways of developing the operational semantics for LIPS. In relation to this, big-step and small-step/Structural Operational semantics are compared. It has been identified that a combination of both the semantics is more suitable to specify the communication and computational part of a distributed programming language like LIPS. The study also suggests that an abstract machine for LIPS can be developed in order to give an intermediate representation for a language. The abstract machine can also be used to implement the compiler of the language.

SACS is a point-to-point message passing system which is an asynchronous variant of SCCS developed to specify the communicating part of LIPS. The main objective of SACS is to separate the specification of communication from the computation part of LIPS programs so that they can proceed independently. As the behaviour of process algebra can be studied using its formal semantics and there is no formal semantics defined for SACS, the work proposes to

1. develop a Structural Operational Semantics for SACS and
2. study the equivalence properties of SACS.

Also, in order to verify the correctness of SACS with its implementation, the study proposes to create an equivalence relation between them. The equivalence relation can be created by using the operational semantics represented as labelled transition system.

The succeeding chapters give an introduction to LIPS and the AMPS, the Asynchronous Message System, implemented in LIPS, and continue to define the formal semantics and specifications for LIPS.

Chapter 3

An Introduction to LIPS and AMPS

LIPS is an asynchronous message passing language which was originally developed to be executed on parallel computers such as transputers. Extensions have been added to make it a distributed programming language in order to take advantage of the available network of personal computers. LIPS has the following characteristics:

- Communication by assignment;
- Separation of communication from computation;
- Asynchronous message passing without buffers;
- Portability.

The asynchronous message passing architecture designed and implemented for LIPS is called the Asynchronous Message Passing System (AMPS). The AMPS is based on a simple architecture comprising of a Data Structure (DS) a Driver Matrix (DM), and interface codes. In a LIPS program, a message is sent and received using simple assignment statements and the program is not concerned with how the data is sent or received. With the network topology and the guarded process definitions, it is easy to identify the variables participating in the message passing using which the DS and the DM are defined. AMPS was conceptualised by [Bavan et al., 2007b]. This work focuses on developing a formal semantics for AMPS [Rajan et al., 2007a] and implementing it in the LIPS compiler.

This chapter gives an overview of LIPS and describes the architecture and working of AMPS. The description of AMPS is taken from [Bavan et al., 2007b].

The chapter is organised as follows:

- Section 3.1 gives an account on the basic elements of a LIPS program.
- Section 3.2 gives an introduction to writing programs in LIPS.
- Section 3.3 explains the architecture of the Asynchronous Message Passing System (AMPS).

- Section 3.4 describes the working of AMPS.
- Section 3.5 demonstrates the working of the AMPS using two case studies.
- Section 3.6 gives a summary.
- Appendix A: Sample LIPS programs which will enable the user to understand the concepts.
- Appendix B: Case study 2 - Post Office Scenario - to demonstrate the working of the AMPS.

3.1 Structure of a LIPS Program

A LIPS program is represented by a network of computational nodes connected by a set of point-to-point unidirectional channels that carry messages between communicating nodes. A LIPS program consists of:

i. a network definition: describes the communication part of the LIPS programs.

ii. nodes definition: describes the computational part of the nodes in the network.

3.1.1 Network Definition

Network definition describes the topology of the network by naming each node (representing a process) and its relationships (in terms of input and output data) to other nodes in the system. In other words, it describes how the nodes are connected through channels.

The only statement which is used to specify the connectivity of nodes in a network is the `connect` statement. This statement specifies input and output(I/O) channels to a node and the name of the node to be executed in that network, in other words, the I/O interface to a particular node.

The syntax for the `connect` statement is as follows:

```
node_label : connect node_name(input_channels) --> (output_channels)
```

An example of a connect statement that links two input channels, a and b, to a node that executes a process P with a node label 7 and produces an output on channel c is shown in Figure 3.1.

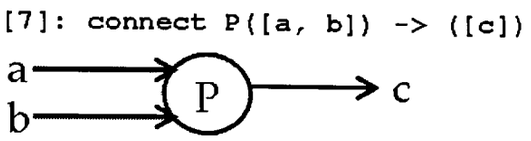


Figure 3.1: Connect process.

Square brackets are used to group data belonging to a particular data type or category. Using the connect statement, networks of any complexity can be built. The connect statements allow the user to fan in and fan out messages. For example, the fanning in and fanning out of the nodes shown in Figure 3.2 is represented using a set of `connect` statements as given below:

```
[1] : connect  U([h]) --> ([i])
[2] : connect  R([i]) --> ([j])
[3] : connect  S([i]) --> ([k])
[4] : connect  T([i]) --> ([l])
[5] : connect  V([j,k,l]) --> ([m])}
```

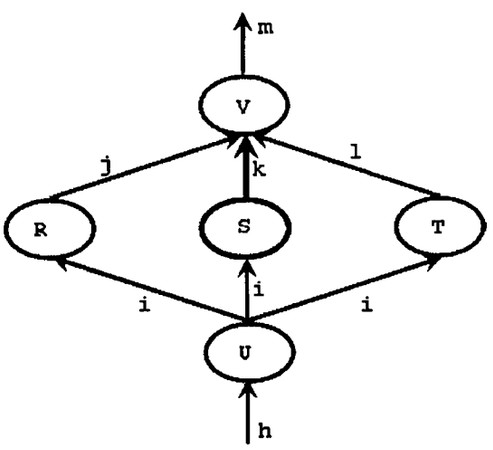


Figure 3.2: Data flow graph illustrating fan-in and fan-out effect via *connect*.

For a given problem, the topology of the network can be shown using the network definition. Not specifying the computational details at the network level has an added advantage as this promotes good design practice by motivating the programmer to produce the target solution model at a higher level of abstraction. Communication is considered as a framework and separated from computation.

3.1.2 Nodes Definition

A LIPS node is a distributable object which can receive or send messages. Among the network of nodes, there is an obligatory host node which is executed first. Each node consists of a set of processes. To execute a process, its precondition/guard needs to be satisfied. Therefore, processes in a LIPS program are called guarded processes. A guard is a list of input channels waiting for the data to be received to activate the guard. The syntax for the node definitions is as follows:

```
Node name(input_channels)  --> (output_channels)
       { variable_declaration;
         guarded_process_1
         ...
         guarded_process_n
       }
```

i. **Channels**

Nodes in a LIPS program communicate with one another by using channels. A channel is a unidirectional link through which messages flow. A channel can

(a) have several endings. For example, consider the network shown in Figure 3.3. Channel x delivers message from node A to nodes B, C, and D.

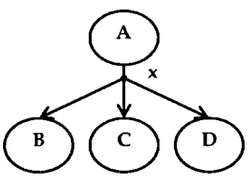


Figure 3.3: Channel with Multiple Outputs

(b) have several beginnings and several endings. For example, consider the network shown in Figure 3.4. Channel y receives a message from either node A or node B but transmits the message to C, D, and E.

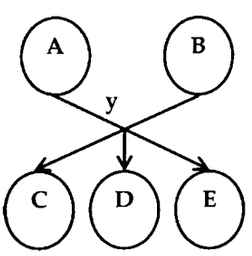


Figure 3.4: Channel with Multiple Inputs

(c) loop back so that the input and output of the channel connect to the same node. This shows the ability of a node to send a message to itself. This is illustrated in Figure 3.5.

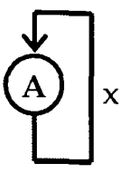


Figure 3.5: Looping Channel

ii. **Guarded Processes**

A process in a node is a guarded process. A guarded process has a guard and a statement block which forms the computational part of LIPS. A guard is a list of input channels. Only when all the input channels of a specific guard receive values, the associated process body will be executed. A guarded process may generate data for the output channels. The output channel is write-only. A situation may occur in which two or more guarded processes may become eligible for execution. When such a condition arises, only one guarded process will be selected for execution. A guarded process has the following syntax:

```
[input_condition] => {statements_block}
```

As described earlier, the input_condition is a set of input channels and all of the input_channels have to receive values for the statements_block to be executed. There are four types of guarded processes:

(a) ***init*** guarded process: This process is optional. When present, it will be executed first automatically at the start of the program. The role of the *init* guarded process is to initialise variables. The *init* guard has no precondition and is represented by an empty pre condition list as shown below:

```
[ ] => {statements_block}
```

(b) ***start*** guarded process: As an initiating guarded process it is obligatory to start the network. There is only one *start* guarded process in a LIPS program. Usually, the host node contains the *start* guarded process. This sends start signals to all the nodes in the network so that they can start their processing. The guard is given a special pre-condition symbol # and the syntax is as follows:

```
[#] => {statements_block}
```

(c) ***conditional*** guarded process: This is optional and has a set of input data channels as its guard which need to be activated for its associated statement_block to be executed. All the input channels in the guard must be true for the guard to be activated. There may be more than one *conditional* guarded processes in a node. Let there be n channels: Ch_1, Ch_2, Ch_3,..., Ch_n. The *conditional* guarded process takes the following form:

$$[Ch_1,\ Ch_2,\ Ch_3,\ldots,Ch_n] \Rightarrow \{statements\_block\}$$

(d) ***locally activated*** guarded process: This is optional and made up of set of input channels. These input channels can be either input channels or local channels to the node. Locally generated message signals are passed via the local channels and their scope is limited to the node that generates them. Let there be n channels: lCh_1, lCh_2, lCh_3,..., lCh_n. The *locallyactivated* guarded process takes the following form:

$$[lCh_1,\ lCh_2,\ lCh_3,\ldots,lCh_n] \Rightarrow \{statements\_block\}$$

Figure 3.6 illustrates the execution sequence of the four types of guards. As far as the

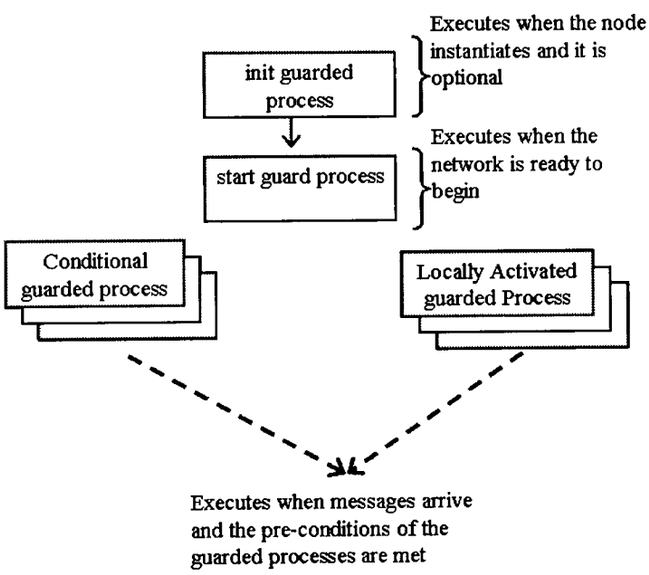


Figure 3.6: Execution Sequence of Guards

statement_block is concerned, it consists of

i. optional variable declarations, and

ii. zero or more statements.

A statement itself is a complete unit of execution. Most of the LIPS statements are typical C programming statements. *stop* and *settimer* are the two additional statements used to stop the execution of the program and to insert a delay in processing respectively.

3.2 Programming LIPS

As discussed in Section 3.1, a LIPS program has two parts: network definition and nodes definition. Consider the example of calculating the area under a curve $y = f(x)$ using Simpson's rule. There are three nodes to be defined: **host, Area, and Summer**. The **host** initialises variables, instantiates other nodes, sends inputs if there are any to other nodes in the network through output channels, and receives values through the input channels. In this example, the **host** sends the *width* of the segments whose *area* is to be calculated and the *segment* numbers to **Area** node. The **Area** node upon receiving the *width* and *segment* numbers calculates the *areas* of the segments and sends them to the **Summer** node. The **Summer** node receives the *areas* of the segments, adds them to find the total area under the given curve and stores in *result*. The *result* is sent to the host.

3.2.1 Programming the Network Definition

Table 3.1 show the input and output channels corresponding to the nodes in the network.

Table 3.1: Input and Output Channel table for the Simpson's rule

Node	Input Channels	Output Channels
host	result	width,segment[0],segment[1],segment[2]
Area	width,segment[0],segment[1],segment[2]	area[0],area[1],area[2]
Summer	area[0],area[1],area[2]	result

A possible network diagram for the Simpson's problem is shown in Figure 3.7.

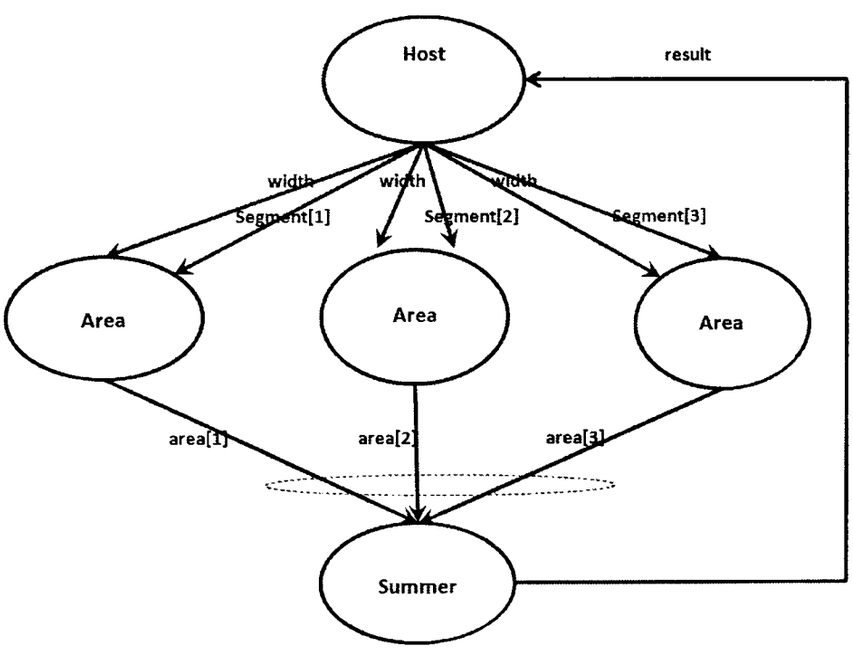


Figure 3.7: Network Diagram for the Simpson's Rule problem

The corresponding network statements for the nodes shown in Table 3.1 are:

```
[1]:connect host ([result]) --> ([width][segment[0 .. 2]]);
[2]:connect Area ([width][segment[0 .. 2]]) --> ([area[0 .. 2]]);
[3]:connect Summer ([area[0 .. 2]]) --> ([result]);}
```

The node identifications used in the connect statements are unique and are used to represent multiple instances of a node precisely and unambiguously.

3.2.2 Programming the Nodes Definition

Having defined the network of a system, this section defines each node with its associated guarded processes. A nodes definition has a header. An example is shown below:

```
Node name(input_channels) -> (output_channels)
```

Following is one possible set of node header definitions for the Simpson's problem:

```
Node host(double result) -> (double width, int segment[2])
Node Area(double width, int segment[2]) -> (double area[2])
Node Summer (double area[2]) -> (double result)
```

The remaining section of this chapter describes one possible code for the host, Area and Summer nodes. The host node has the init, start and conditional guarded processes. The init guarded process initialises the values for width and segment.

```
//init guarded process
[ ] => //send the width and segment to Area
  {
      int i;
      width = .333;
      for (i=0; i<=2; i++){
          segment[i] = i+i;
      }
  }
```

In the current scenario, the *start* guarded process has no executable statements but this is required to start the network and the code segment is given below:

```
//start guarded process
[#] => //send the start signal to other nodes
  {
     // start the process
  }
```

The *init* and *start* guarded processes are obligatory and executed once. The *conditional* and *locallyactivated* guarded processes may be executed more than once when their pre-condition is met, i.e, when their input channels carry new values. There is one *conditional* guarded process defined for the host node to receive the *result* and represented as:

```
//conditional guarded process
[result] => //receives the result - the area under the given curve
  {
     print("The area under the given curve is = ", result);
  }
```

Similarly **Area, and Summer** nodes can be defined using *conditional* guarded processes. One possible set of codes for each of the nodes is shown below.
conditional guarded process for the **Area** node:

```
[segment[0..2], width] =>
  {
      //receives segment numbers and width and calculates area
      int j;
      for(j=0; j<=2; j++){
          x = width * (2.0 * segment[j] + 1.0) / 2.0;
          y = 4.0 / (1.0 + (x * x));
          area[j] = x * y; //outputs sent to output channels
  }
```

conditional guarded process for the **Summer** node:

```
[area[0..2]] =>
  {
      //receives areas for the segments and calculates total area
      double total;
      int count;
      for(count=0; count<=2; count++){
          total = total + area[count];
          result = total; //outputs sent to output channels
  }
```

The complete program to calculate the area under a curve using Simpson's rule is given in Appendix A.1.

3.2.3 Compiling and Running a LIPS program

The LIPS compiler has been developed using *JFlex*, the Java LEXical analyser, written in Java and CUP, which is a Java based Constructor of Useful Parsers. *CUP* is similar

to the widely used YACC. It is a *LALR* (Look_Ahead Left to Right) parser generator. *JFlex* is generated to work together with the *LALR* parser generator *CUP*.

The compiler takes the native LIPS code and generates the Java code. This Java code can then be compiled using a Java compiler. Java compiler version jdk1.5.0_01 has been used. The main purpose of developing the compiler was to test the asynchronous message passing using AMPS. The compiler developed so far has been tested with simple applications. It is necessary to do rigorous testing with more complex systems but this is not included in this research. The AMPS of LIPS is described in the subsequent sections.

3.3 Architecture of the Asynchronous Message Passing System (AMPS)

The AMPS of LIPS consists of a very simple Data Structure (DS) and a Driver Matrix (DM). The compiler of LIPS automatically generates the DS, DM and the interface codes necessary for the AMPS of LIPS. This section describes the DS and the DM used for message passing.

3.3.1 The Data Structure of the AMPS

The Data Structure (DS) is a linked list where all the nodes in the network including the host node are linked to all the other nodes. Each node has the following six components:

1. A node number (**NodeNum**- *an integer*) - a unique number is assigned to each node;
2. Name of the function it is executing (**name** - *a symbolic name*);
3. A pointer to the next node in the system;
4. Two more pointers. For the node under consideration:
   (a) A list of associated input channel variables;
   (b) A list of associated output channel variables.
5. Each channel variable consists of a data field giving the channel number (**vnum** - *an integer*) and two pointers, one pointing to the next channel variable in the list and the other points to a record.

   The record of the input channel contains:

   (a) Channel name (**var1** - *symbolic name*). This is used for debugging purposes;

(b) Currency of the data - old data (status = 0) or new data (status = 1) present. Only data with status = 1 is passed on to a node for processing;

(c) Value of the data. (**value** - *actual value of specified type*).

The record of the output channel contains:

(a) Channel name (**var1** - *symbolic name*);

(b) The number of nodes that are to receive the data (**counter** - *1..n*), which will be decremented as a copy of the data is transferred to a destination node. New data is only accepted *written* when the counter is 0;

(c) Value of the data (**value** - *actual value of specified type*).

Figure 3.8 depicts the data structure of the AMPS. It is assumed that there are **p** number of **IN VECTORS** and **m** number of **OUT VECTORS** where **p** and **m** are positive integers.

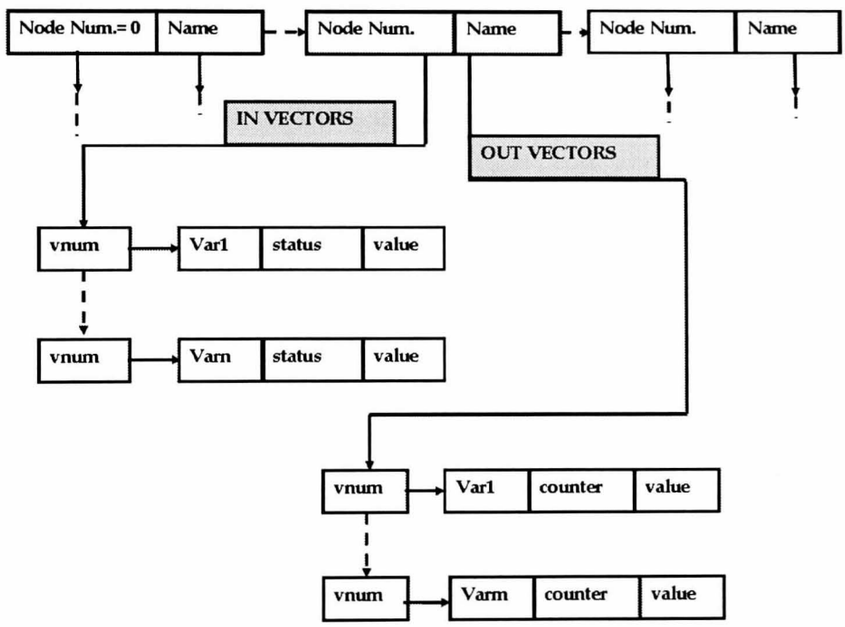


Figure 3.8: Data Structure of the AMPS.

3.3.2 The Driver Matrix of the AMPS

The Data Matrix *DM* facilitates the distribution of messages. The structure of DM is shown in Figure 3.9. The DM contains the details of channel variables in the network as described below:

1. The channel number, (**vnum**- *an integer*);

2. The node number (source node) from where the channel variable originates (**SrcNodeNum**- *an integer*);

Source Nodes			Destination Nodes						
Vnum	Srcnd	type	Nd0	Nd1	Nd2	:	:	Nd9	Nd10
0	0	3	0	0	0	:	:	0	0
1	1	2	0	0	1	:	:	1	1
2	3	4	0	1	0	:	:	0	0
:	:	:	:	:	:	:	:	:	:
:	:	:	:	:	:	:	:	:	:
9	3	1	1	0	0	:	:	0	0
10	4	1	1	0	0	:	:	0	0

Figure 3.9: Data Structure of the AMPS.

3. The data type of each channel variable (**type** - *integer*);

4. The nodes where they are sent as input and the destination nodes (either '1' or '0' in the appropriate column). A '1' in a column indicates that the corresponding destination receives a copy of the input or a '0' (otherwise).

All the values in the matrix are integers. The integer values given to the source and destination nodes are the same as the node numbers used in the DS. The following section describes the working of the AMPS.

3.4 The Operation of the AMPS

When a node outputs a message, a message packet in the following format is generated:

SrcNodeNum	vnum	type	data

Message packet-1 sent from a node

Once a piece of data is ready, the process makes the following call to the AMPS.

Is_ok_to_send(SrcNodeNum, vnum)

When this call is received, the AMPS checks the DS to see if the output channel of the node has its counter set to zero. If it is set to zero, it returns a value 1 else 0.

- If 0 is received, the sending process waits in a loop until 1 is received.
- If 1 is received, the sender node sends the message in a packet using the following call:

Send(SrcNodeNum, vnum, type, data)

On the receipt of this packet, the AMPS checks the DS to see whether the **vnum** and **type** are correct and stores the data in the appropriate field. The counter is set to the number of nodes that are to receive the data by consulting the DM. The **Send** function returns a 1 to indicate a success.

After storing the data, the AMPS consults the DM, distributes the data to other DS nodes, and decrements the counter accordingly. Here the data is written to the input channel variable of a receiving DS node, provided the status of that input channel variable is 0 (that is, the channel is free to receive new data). Once the data is received, the status is set to 1. If any of the DS destination nodes are unable to receive the new data, the AMPS periodically checks whether they are free to accept the data. No new value will be accepted for the output channel from where the data is being distributed until the counter becomes 0.

When a guard in a node requires an input, it makes the following call to the AMPS:

Is_input_available(NodeNum, vnum)

The AMPS checks the appropriate DS node and the channel variable number, **vnum**. If the status is 1, the function returns a 1 to tell the caller with the node number **NodeNum** that the data is available, else it returns a 0.

- If a 1 is returned then the node makes a request to the AMPS to send the data. The AMPS extracts the data from the appropriate channel of the DS, returns it to the calling process, and sets the status to 0. The data is sent in the following format:

NodeNum	vnum	type	data

Message packet-2 sent from the AMPS

- If a 0 is returned to the **Is_input_available** function, then the process continues processing or repeats the call.

3.5 Case Studies

The AMPS system has been implemented in the LIPS compiler. Test results have shown that the AMPS passes messages in an asynchronous fashion effectively across different platforms without any message buffers. This section demonstrates the working of the AMPS in LIPS using two case studies. First, the vending machine problem, is discussed in the next subsection and the second, the post office scenario, is described in Appendix B.

3.5.1 Case Study 1: Vending Machine Problem

Consider a vending machine that requires a CUSTOMER to insert a `coin` and press a `button`, after which the machine will serve a drink. We have split the vending machine into MACHINE_INTERFACE, and MACHINE_INTERNALS. When MACHINE_INTERFACE receives the `coin` and `button`, it generates the `drkSig`.

There is a process called INIT which outputs `trayEmpty` signal to infer the MACHINE_INTERNALS that the tray is empty. Without such a signal, the vending machine would accept the `coin` and `button` press and deliver the drink without checking whether previously delivered drink has been removed. This may result in a vending machine that delivers drink one above the other. MACHINE_INTERNALS will `deliver` the drink only after receiving the `trayEmpty` signal from INIT and `drkSig` from the MACHINE_INTERFACE.

This means that there are four processes involved in modelling the vending machine: `INIT (host)`, `CUSTOMER`, `MACHINE_INTERFACE`, `MACHINE_INTERNALS`. A diagrammatic representation of the vending machine problem is shown in Figure 3.10.

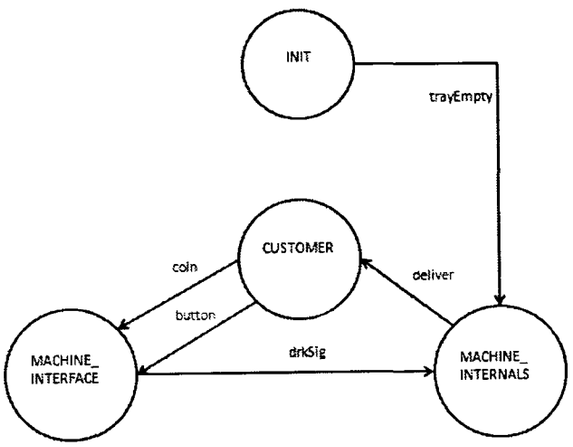


Figure 3.10: Vending Machine.

When the LIPS program for the vending machine problem is compiled, the compiler will generate the Driver Matrix(DM) and the Data structure (DS) needed for message passing. The DS is initialised with `null` values and is shown in Figure 3.11. Its corresponding DM is shown in Table 3.2.

Table 3.2: Driver Matrix for the Vending Machine Problem.

vnum	SrcNodeNum	type	Destination nodes			
0	1	4	0	0	0	1
1	2	1	0	0	1	0
2	2	4	0	0	1	1
3	3	4	0	0	0	1
4	4	4	0	1	0	0

```
Data Structure of Vending Machine Problem

1       Host
input list finished
0       trayEmpty       0       null
output list finished
        2       Customer
4       deliver 0       null
input list finished
1       coin    0       null
2       button  0       null
output list finished
        3       Mac_Interface
1       coin    0       null
2       button  0       null
input list finished
3       drkSig  0       null
output list finished
        4       Mac_Internal
0       trayEmpty       0       null
3       drkSig  0       null
input list finished
4       deliver 0       null
output list finished
```

Figure 3.11: Data Structure for the Vending Machine Problem.

The DS will be updated whenever data is received and sent. At any point of time, the DS shows the up-to-date information of the AMPS.

3.6 Summary

This chapter described the LIPS programming language and presented a simple asynchronous message passing system that is used in LIPS.

LIPS offers distinct advantages in the programming of parallel and distributed systems.

especially in the area of communication and avoidance of deadlock and livelock as described in [Bavan and Illingworth, 2001]. The compiler of LIPS has been developed based on its high level specification and operational semantics.

The Asynchronous Message Passing System (AMPS) proposed by Bavan et al. [2007b] has been implemented into the LIPS compiler. The significant feature of AMPS is that it does not use any buffers. Instead it uses two storage spaces for each data item transferred. One storage space is held by the data structure, DS, and the other is by the process that generates the data. The process of distributing the messages is driven by a driver matrix, DM, which is essentially a bit map that helps to reduce memory usage. AMPS is presented as a centralised system to aid understanding but it can be partitioned appropriately and placed in different processors to make the message passing more efficient and localise the inter-process communication within a processor. Considering the fact that there is no loss of data and the overhead involved is minimal, the small delay incurred due to lack of buffers is acceptable. AMPS has been initially developed for LIPS but can be used for other language systems with minor alterations. The formal semantics for the AMPS [Rajan et al., 2007a] has been described in Section 4.3 of Chapter 4.

Chapter 4

Operational Semantics for LIPS

This chapter describes the work done on developing a formal definition of the operational semantics and abstract machine of LIPS. As discussed in the Literature Review (Chapter 2), semantics of any programming language can be represented in many ways. In order to adequately provide implementation information for both computational and communication parts of LIPS, we follow a two step strategy:

- Firstly, big-step semantics has been used to define the computations in a LIPS program. Big-step semantics has been chosen due to its capability to describe the computations by providing a direct relation between initial and final states and hiding the internal steps of evaluation.

- Secondly, we extend the big-step semantics with Structural Operational Semantics (SOS) to describe the asynchronous communication of LIPS. SOS has been chosen to describe the communication in opposite to a big step semantics as it will tell us how the intermediate steps of the execution are performed which are crucial in message passing.

The combined semantics describes the operational behaviour of LIPS programs by modelling how different statements are executed while capturing both the result of computation and how the result is produced. This can help to implement the language and its debugging tools.

The particular style of operational semantics used for the computational part of LIPS and the abstract machine of LIPS was inspired by Crole (2006). His definition uses an evaluation relation to describe the operational semantics to show how an expression evaluates to a result to yield a change of state and a compiled Code Stack State (CSS) machine for an IMPerative experimental language called IMP.

Abstract machines have been used as low-level architectures suitable for supporting implementations of a wide variety of programming languages, including imperative, functional, and logic programming languages [Hannan and Miller, 1992]. They are distinguished

from operational semantics as they provide intermediate representation of the language's implementation. An abstract machine called the LIPS Abstract Machine (LAM) has been defined to execute LIPS programs.

LAM works on the principle of single-step re-write rules describing single-step operation on the state of the computation. Re-write rules are used as they explicitly show individual steps of execution and provide an intermediate level of representation for many practical implementations of programming languages [Hannan and Miller, 1992]. The correctness of the execution of the LIPS program/expression written using the operational semantics is verified by comparing the result with the result of executing the same code written using the LAM of LIPS.

The operational semantics described for the computational part of LIPS has been extended to accommodate the communication part of LIPS and is implemented using the following:

- `connect` statements which define the topology of the network (described in Section 3.1.1 of Chapter 3).
- Node which contains the set of guarded processes (described in Section 3.1.2 of Chapter 3).
- Asynchronous Message Passing System (AMPS) (described in Section 3.3 of Chapter 3).

The work described on the operational semantics for the communication part of LIPS is a refinement of the work published in [Rajan et al., 2007a].

The chapter has been organised as follows:

- Section 4.1: describes the operational semantics for the computational part of LIPS.
- Section 4.2: defines the LAM, the LIPS Abstract Machine.
- Section 4.3: describes the operational semantics for the communication part of LIPS.
- Section 4.4: gives the necessary re-write rules and the LAM code for the communication part of LIPS.
- Section 4.5: summary.

4.1 Operational Semantics for the Computational Part of LIPS

The Computational part of a LIPS program comprises of the statement blocks associated with the guarded processes. The operational semantics of the statements in the statement block are described by listing

- the syntactic categories,
- the type assignments, and
- the evaluation relation which specifies the start state of the program, its transition semantics, and the final result.

Operational semantics operate on the abstract syntax of the programming language. The evaluation relation

$$(P, Q) \Downarrow (P, S')$$

is created using the abstract syntax. This section defines the following:

- the abstract syntax for the computational part of LIPS,
- its type system which is denoted as

$$P : : \sigma \text{ (the program expression P is of type } \sigma), \text{ and}$$

- the operational semantics described using an evaluation relation.

The configuration of the instruction in the statement block of the guarded processes consists of a program expression at a specified state. A Program expression in LIPS is a combination of values, variables, operators, and commands. A detailed description about the expression used in a LIPS program can be found in Section 4.1.1 and the list of LIPS expressions can be found in Table 4.5. If the program expression under consideration is a command, it is described using a set of computations to result in a change of state.

4.1.1 Abstract Syntax for the Computational Part of LIPS

As a first step in defining the abstract syntax, we list the syntactic categories in Table 4.1.

Table 4.1: Syntactic Categories for the Computational Part of LIPS

Set of integers	$\mathtt{Z} \stackrel{def}{=\!=} \{..., -1, 0, 1, ...\}$
Set of real numbers	$\mathtt{R} \stackrel{def}{=\!=} \{decimals\}$
Set of Booleans	$\mathtt{B} \stackrel{def}{=\!=} \{true, false\}$
Set of strings	$\mathtt{STR} \stackrel{def}{=\!=} \{stringliterals\}$
Set of character symbols	$\mathtt{C} \stackrel{def}{=\!=} \{charaterlierals\}$
Set of locations	$\mathtt{LOC} \stackrel{def}{=\!=} \{L_1, L_2, ...\}$
Integer constant	$\mathtt{ICst} \stackrel{def}{=\!=} \{\underline{n} \mid n \in \mathtt{Z}\}$
Real number constant	$\mathtt{RCst} \stackrel{def}{=\!=} \{\underline{n} \mid n \in \mathtt{R}\}$
String constant	$\mathtt{SCst} \stackrel{def}{=\!=} \{\underline{str} \mid str \in \mathtt{STR}\}$
Character constant	$\mathtt{CCst} \stackrel{def}{=\!=} \{\underline{char} \mid char \in \mathtt{CHAR}\}$
Boolean constant	$\mathtt{BCst} \stackrel{def}{=\!=} \{\underline{b} \mid b \in \mathtt{B}\}$
Fixed, finite set of numeric operators	$\mathtt{NOpr} \stackrel{def}{=\!=} \{+, -, /, *, ++, --, +=, -=, *=\}$
String operator	$\mathtt{SOpr} \stackrel{def}{=\!=} \{+\}$
Character operator	$\mathtt{COpr} \stackrel{def}{=\!=} \{+\}$
Fixed, finite set of Boolean operators	$\mathtt{BOpr} \stackrel{def}{=\!=} \{=, <>, <, <=, >=, >, \&\&, \|\|, \sim\}$

Let $\underline{c}$ be a constant range over the elements $\mathtt{Z} \cup \mathtt{R} \cup \mathtt{B} \cup \mathtt{STR} \cup \mathtt{CHAR}$ and l over $\mathtt{LOC}$. All the operators except '+' are regarded as mathematical operators. '+' behaves as a mathematical operator only when it is used on numeric pairs and it behaves as a concatenation operator, when it is used on a pair of strings or characters. With these assumptions, the set of LIPS expression constructors is specified as follows:

$$\mathbf{Loc} \cup \mathbf{ICst} \cup \mathbf{BCst} \cup \mathbf{SCst} \cup \mathbf{CCst} \cup \mathbf{NOpr} \cup \mathbf{BOpr} \cup \mathbf{SOpr} \cup \mathbf{COpr} \cup \{the\ set\ of\ LIPS\ commands\}.$$

Let Table 4.2 list the set of operators used in program expressions.

Table 4.2: Set of Operators of LIPS

nop $\in$ **NOpr**
bop $\in$ **BOpr**
sop $\in$ **SOpr**
cop $\in$ **COpr**

The operator op ranges over **NOpr** $\cup$ **BOpr** $\cup$ **SOpr** $\cup$ **COpr**. The types of LIPS statements and their purposes are listed in Table 4.3. We refer to them as expressions. Both `if` and `switch` are symbolised as `if-command` and can be defined using a single specification. Similarly, `for`, `do-while` and `while` can be defined using a single specification.

Let P denote the elements of program expressions. With the above assumptions and definitions, the set of expressions, Exp, of LIPS programs can be defined inductively as shown in Table 4.4.

Table 4.3: LIPS Statements/Expressions

Group Name	Statements	Purpose
Simple Statements	Expression	Assigns a value to a variable
	Empty	Does Nothing
	Read	Reads input variables
	Print	Displays values on the console
	Break	Exits the compound block
	Stop	Exits the program
	Continue	Restarts loop
	Timer	Introduces delay
Branch Statements	if	Decision making
	switch	Decision making
Loop Statements	for	Iterate over a range of values
	do-while	Iterate a block of statements while a Boolean expression remains true
	while	Iterate a block of statements while a Boolean expression remains true

Table 4.4: Set of Operators of LIPS

P nop P' is to be used for the finite tree as `nop(P, P')`
P bop P' is for $bop(P, P')$
P sop P' is for $sop(P, P')$
P cop P' is for $cop(P, P')$
if P then P' else P" is for $cond(P, P', P'')$ and
while P do P' for $while(P, P')$

Each of the program expressions is a finite tree whose nodes are labelled with constructors. The set of abbreviations known as syntactic sugar[1] of LIPS shown in Table 4.4 is formed based on

- set of operators in Table 4.2,
- the Exp of LIPS expressions in Table 4.5,
- the LIPS command set shown in Table 4.6.

[1]Syntactic sugar gives the designer, in the case of specification computer languages an alternative way of specifying that is more practical, either by being more succinct or more like some familiar notation. It does not affect the expressiveness of the formalism (From Wikipedia, the free encyclopedia & Landin,P.J. August 1965, A generalisation of jumps and labels, Report, UNIVAC systems Programming Research)

Table 4.5: Exp of LIPS Program Expressions

P ::=		
	$\underline{c}$	constant
|	l	memory location
|	nop(P, P')	arithmetic operator
|	bop(P, P')	Boolean operator
|	sop(P, P')	String operator
|	cop(P, P')	character operator
|	empty	do nothing
|	break	break the current loop
|	stop	stop execution
|	continue	restart loop
|	timer	delay process
|	assign(l, P')	assignment
|	sequence(P, P')	sequencing
|	read($l_1, l_2, ..., l_n$)	read input variables
|	print(P_1|$\underline{c_1}$,P_2|$\underline{c_2}$,...,P_n|$\underline{c_n}$)	print values
|	cond(P,P',P")	conditional
|	while(P, P')	while loop

When evaluating the expressions, a set of priority rules is applied. They are:

- The arithmetic, string, and character operators are always grouped to the left. Let P_1 op P_2 op P_3 be the arithmetic expression under consideration. This will be evaluated as (P_1 op P_2) op P_3.

- While arithmetic operators follow the precedence rules of Java language, there is no priority rule required for `SOpr` and `COpr` which stand for String Operator and Character Operator respectively.

- The sequencing associates to the right. This means that in the case of an assignment expression, the right hand side of an assignment expression is evaluated. For example, this clause allows i = i + 1.

- Brackets are used as informal punctuations while writing expressions. Consider the syntax trees of "`if P then` P_1 `else` (P_2;P_3)" and "`(if P then` P_1 `else` P_2);P_3" in Figure 4.1 to understand sequencing with brackets.

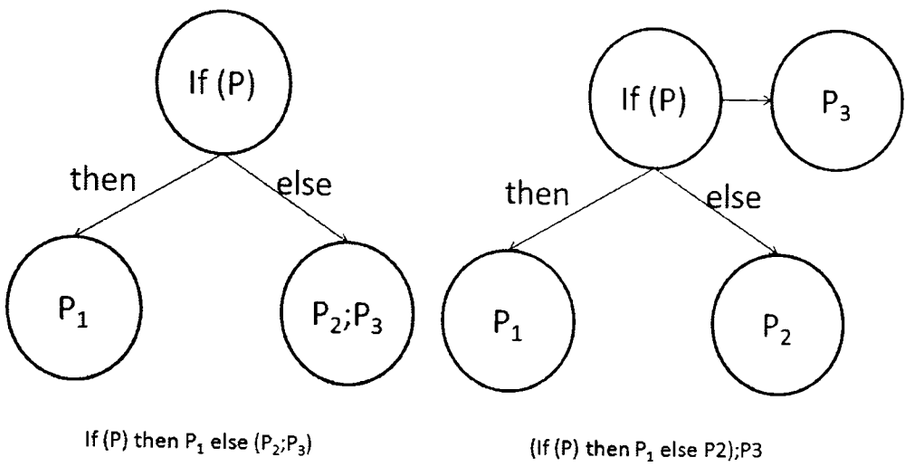


Figure 4.1: Syntax Trees for "`if P then P1 else (P2;P3)`" and "`(if P then P1 else P2);P3`".

4.1.2 Types in LIPS

Definition 1. The types in a language include both data types and commands that evaluate to values of specific data types. The types in LIPS are given by the grammar:

$$\sigma ::= \texttt{int} \mid \texttt{real} \mid \texttt{bool} \mid \texttt{string} \mid \texttt{char} \mid \texttt{cmd}$$

`cmd` is the list of commands/expressions.

If a program expression P is of type σ, we specify that as:

$$P :: \sigma$$

and this statement is known as a type statement. There is no reference made to variables in the abstract machine. Memory location l is used to store data of a specific type. Let L, the location environment, be a finite set of location pairs, (l, σ), where l is the location and σ is the type.

The location of each type is required to be unique. Let n be the number of locations to be used for integer type, m number of locations to be used for real/float, s number of locations for strings, t number for Boolean, and r for character. We specify the typical location environment as follows:

$$\begin{aligned} \mathrm{L} = \{ & l_1 :: \texttt{int}, ..., l_n :: \texttt{int}, \; l_{n+1} :: \texttt{real}, ..., l_{n+m} :: \texttt{real}, \\ & l_{n+m+1} :: \texttt{string}, ..., l_{n+m+s} :: \texttt{string}, \\ & l_{n+m+s+1} :: \texttt{bool}, ..., l_{n+m+s+t} :: \texttt{bool}, \\ & l_{n+m+s+t+1} :: \texttt{char}, ..., l_{n+m+s+t+r} :: \texttt{char}\} \end{aligned}$$

The LIPS type assignment statements are defined inductively using the rules shown in Table 4.6. Let 'any' specify any type of data which may be useful while forming Boolean expressions.

Table 4.6: Type Assignments P :: σ of LIPS

$\frac{}{\underline{n} :: \texttt{int}}\ [\forall n \in \texttt{Z}] :: \texttt{INT}$	$\frac{}{l : \texttt{int}}\ l :: \texttt{int} \in \texttt{L} :: \texttt{INTLOC}$
$\frac{}{\underline{r} :: \texttt{real}}\ [\forall r \in \texttt{R}] :: \texttt{REAL}$	$\frac{}{l : \texttt{real}}\ l :: \texttt{real} \in \texttt{L} :: \texttt{REALLOC}$
$\frac{}{\underline{str} :: \texttt{string}}\ [\forall str \in \texttt{STR}] :: \texttt{STRING}$	$\frac{}{l : \texttt{string}}\ l :: \texttt{string} \in \texttt{L} :: \texttt{STRINGLOC}$
$\frac{}{\underline{T} :: \texttt{bool}} :: \texttt{TRUE}\quad \frac{}{\underline{F} :: \texttt{bool}} :: \texttt{FALSE}$	$\frac{}{l : \texttt{bool}}\ l :: \texttt{bool} \in \texttt{L} :: \texttt{BOOLLOC}$
$\frac{P_1 :: \texttt{int}\ P_2 :: \texttt{int}}{P_1\ \texttt{nop}\ P_2 :: \texttt{int}}[\texttt{nop} \in \texttt{NOpr}] :: \texttt{NOP}$	$\frac{P_1 :: \texttt{real}\ P_2 :: \texttt{real}}{P_1\ \texttt{nop}\ P_2 :: \texttt{real}}[\texttt{nop} \in \texttt{NOpr}] :: \texttt{NOP}$
$\frac{P_1 :: \texttt{string}\ P_2 :: \texttt{string}}{P_1\ \texttt{sop}\ P_2 :: \texttt{string}}[\texttt{sop} \in \texttt{SOpr}] :: \texttt{SOP}$	$\frac{P_1 :: \texttt{char}\ P_2 :: \texttt{char}}{P_1\ \texttt{cop}\ P_2 :: \texttt{char}}[\texttt{cop} \in \texttt{COpr}] :: \texttt{COP}$
$\frac{P_1 :: \texttt{any}\ P_2 :: \texttt{any}}{P_1\ \texttt{bop}\ P_2 :: \texttt{bop}}[\texttt{bop} \in \texttt{BOpr}] :: \texttt{BOP}$ where any may be NOP, SOP, BOP, COP	$\frac{}{\texttt{empty} :: \texttt{cmd}} :: \texttt{EMPTY}$
$\frac{}{\texttt{break} :: \texttt{cmd}} :: \texttt{BREAK}$	$\frac{}{\texttt{stop} :: \texttt{cmd}} :: \texttt{STOP}$
$\frac{l :: \sigma\ P :: \sigma}{l := P :: \texttt{cmd}}\ [\sigma] ::$ ASGNMNT where σ is int or real or string or char or bool	$\frac{P_1 :: \texttt{cmd}\ P_2 :: \texttt{cmd}}{P_1\ ;\ P_2 :: \texttt{cmd}} :: \texttt{SEQ}$
$\frac{l_1 :: \sigma\ l_2 :: \sigma\ \ldots\ l_n :: \sigma}{read\ l_1, l_2, \ldots l_n :: \texttt{cmd}}\ [\sigma] ::$ READ where σ is int or real or string or char or bool	$\frac{P_1 :: \texttt{cmd}\ P_2 :: \texttt{cmd}\ \ldots\ P_n :: \texttt{cmd}}{print\ P_1\ ;\ P_2\ ; \ldots ; P_n :: \texttt{cmd}} :: \texttt{PRINT}$
$\frac{P_1 :: \texttt{bool}\ P_2 :: \texttt{any}\ P_3 :: \texttt{any}}{if\ P_1\ then\ P_2\ else\ P_3 :: \texttt{cmd}}$	$\frac{P_1 :: \texttt{bool}\ P_2 :: \texttt{any}}{while\ P_1\ do\ P_2 :: \texttt{cmd}}$

Few examples are listed below to show the deduction of type assignments.

Example 1. Let $L_1 :: real$ be a location environment L which occupies the first location of memory. The deduction for the LIPS expression

$$L_1 = (L_1 + \underline{5.0}) * (L_1 + \underline{7.67})$$

is given below:

$$\frac{L_1 :: REAL \; D_1 \; D_2}{L_1 := (L_1 + \underline{5.0}) * (L_1 + \underline{7.67})} :: cmd :: ASGNMNT$$

$$D_1 \stackrel{def}{=\!=} \frac{\overline{L_1 :: real} :: REAL \quad \overline{\underline{5.0} :: real} :: REAL}{L_1 + \underline{5.0} :: real} :: NOP$$

$$D_2 \stackrel{def}{=\!=} \frac{\overline{L_1 :: real} :: REAL \quad \overline{\underline{7.67} :: real} :: REAL}{L_1 + \underline{7.67} :: real} :: NOP$$

Example 2. Let $L_1 :: int$ and $L_2 :: int$ be the two pairs in the location environment L which occupy the first two consecutive locations of memory. Consider the following while expression:

$$while \; L_1 \leq 1 \, do (if \; L_1 = 1 \; then \; L_2 := 1 \; else \; (L_1 = L_1 - 1; L_2 = L_2 * L_1)) :: cmd$$

This while expression contains a conditional expression

$$if \; L_1 = 1 \; then \; L_2 := 1 \; else \; (L_1 = L_1 - 1; L_2 = L_2 * L_1)$$

and a sequence of expressions

$$L_1 = L_1 - 1; L_2 = L_2 * L_1$$

The deduction for the while statement can be given as follows:

$$\frac{D_1 \dfrac{D_2 \; D_3 \dfrac{D_4 \; D_5}{L_1 = L_1 - 1; L_2 = L_2 * L_1} :: SEQ}{if \; L_1 = 1 \; then \; L_2 := 1 \; else \; (L_1 = L_1 - 1; L_2 = L_2 * L_1)} :: COND}{while \; L_1 \leq 1 \, do (if \; L_1 = 1 \; then \; L_2 := 1 \; else \; (L_1 = L_1 - 1; L_2 = L_2 * L_1))}$$

$$D_1 \stackrel{def}{=\!=} \frac{\overline{L_1 :: int} :: INT \quad \overline{\underline{1} :: int} :: INTLOC}{L_1 \leq \underline{1} :: bool} :: BOP$$

$$D_2 \stackrel{def}{=\!=} \frac{\overline{L_1 :: int} :: INT \quad \overline{\underline{1} :: int} :: INTLOC}{L_1 = \underline{1} :: bool} :: BOP$$

$$D_3 \stackrel{def}{=\!=} \frac{\overline{L_2 :: int} :: INT \quad \overline{\underline{1} :: int} :: INTLOC}{L_2 = \underline{1} :: bool} :: ASGNMNT$$

$$D_4 \stackrel{def}{=\!=} \frac{\overline{L_1 :: int} :: INT \quad \overline{L_1 :: int} :: INT \quad \overline{\underline{1} :: int} :: INTLOC}{L_1 = L_1 - \underline{1} :: int} :: ASGNMNT$$

$$D_5 \stackrel{def}{=\!=} \frac{\overline{L_2 :: int} :: INT \quad \dfrac{\overline{L_2 :: int} :: INT \quad \overline{L_1 :: int} :: INT}{L_2 * L_1 :: int} :: NOP}{L_2 = L_2 * L_1 :: int} :: SEQ$$

In this section, we define the type assignments used in the computational part of LIPS. In the next section, the operational semantics using its evaluation relation is defined.

4.1.3 Operational Semantics for the Computational part of LIPS

The operational semantics for a statement block associated with a guarded process is described using natural/big-step semantics. As stated in the beginning of this chapter, big-step semantics has been chosen as it is simple and easy to implement, describes how the evaluation of expressions and statements affects the program state, and, in the case of an expression, what is the resulting value [Strecker, 2002]. It uses an evaluation relation which explains what is to happen when a program expression at a specific state is executed. The evaluation relation takes the following form:

$$(P, s) \Downarrow (P', s')$$

The above implies that P evaluates to P' and resulting in change of state from s to s'. P can take any of the types discussed in Section 4.1.2.

Definition 2. A state s is a partial function which maps a set of locations to any data type.

$$\text{Loc} \rightarrow \mathbf{Z} \cup \mathbf{R} \cup \mathbf{B} \cup \text{STR} \cup \text{CHAR}$$

If s $\in$ States and $l \in$ Loc and $s(l)$ is defined, then $s(l)$ is the data held in location l at state s.

Example 3. Consider the following:

$$\text{s} = \{l_1 \mapsto 6,\ l_2 \mapsto 56.4,\ l_3 \mapsto TRUE,\ l_4 \mapsto \text{"}apple\text{"},\ l_5 \mapsto 's'\}$$

which means
$s(l_1) = 6$; $s(l_2) = 56.4$; $s(l_3) = TRUE$; $s(l_4) =$ "$apple$"; $s(l_5) = 's'$

The general finite state takes the following form:

$$\text{s} = \{l_1 \rightarrow c_1,\ l_2 \rightarrow c_2,\ \ldots l_n \rightarrow c_n\} \text{ where } n \text{ is a positive integer.}$$

A LIPS expression, P, can be evaluated only when it is of type `int|real|string|bool|char|cmd`. The following assertions are used while defining the operational semantics for the LIPS expressions:

- $(P, s) \Downarrow (\underline{n}, s)$ means that expression P evaluates to a number $n \in \mathbf{Z}|\mathbf{R}$ with no change of state.
- $(P, s) \Downarrow (\underline{str}, s)$ means that expression P evaluates to a string $s \in \mathbf{STR}$ with no change of state.
- $(P, s) \Downarrow (\underline{char}, s)$ means that expression P evaluates to a string $char \in \mathbf{CHAR}$ with no change of state.
- $(P, s) \Downarrow (\underline{bool}, s)$ means that expression P evaluates to a string $s \in \mathbf{B}$ with no change of state.

Definition 3. If $s \in States$, $l \in Loc$, and the constant $\underline{c} \in \mathbf{Z} \cup \mathbf{R} \cup \mathbf{B} \cup \mathtt{STR} \cup \mathtt{CHAR}$ then s is updated with constant $\underline{c}$ assigned to location l. This is written as:

$$s = \{l \rightarrow \underline{c}\}$$

There exists a partial function $s = \{l \rightarrow \underline{c}\} : Loc \rightarrow \mathbf{Z} \cup \mathbf{R} \cup \mathbf{B} \cup \mathtt{STR} \cup \mathtt{CHAR}$ for each $l \in Loc$ which can be defined as:

$$(s = \{l \rightarrow \underline{c}\})(l') \stackrel{def}{=\!=} \begin{cases} \underline{c} & if\ l' = l \\ s(l') & otherwise \end{cases}$$

Definition 4. The set $\Downarrow$ of LIPS configurations can be inductively defined by the set of evaluation rules where

$$\Downarrow \subseteq (Exp \,\mathtt{X}\, States)\,\mathtt{X}\,(Exp \,\mathtt{X}\, States)$$

The rules are formed based on the assumption that $s \in States$ and $l \in Loc$ and $s(l)$ is the data held in location l at state s.

The list of evaluation rules with their deduction tree specifications are given below:

Rule 1: A memory location holding a value refers to the data held in it with no change of state.

$$\frac{}{(l, s) \Downarrow (s(l), s)}\,[provided\ l \in\ domain\ of\ s] \Downarrow LOC$$

Rule 2: A constant value evaluates to itself with no change of state.

$$\frac{}{(\underline{c}, s) \Downarrow (\underline{c}, s)} \Downarrow CONST$$

Rule 3: For any program expressions P_1 and P_2 of a specific data type and an operator (*nop*, *sop*, *cop*, *and bop*) which can be operated on the given program expressions, the evaluation rule is given by

- first evaluating the two program expressions to yield constant values $\underline{n_1}$ and $\underline{n_2}$ and
- then applying the operator between the two resultant values with no change of state.

The deduction tree for integer, real, string, character and Boolean expressions are represented below:

$$Integer|RealExpression \stackrel{def}{=\!=} \frac{(P_1,\ s) \Downarrow (\underline{n_1},\ s)\ (P_2,\ s) \Downarrow (\underline{n_2},\ s)}{(P_1\ nop\ P_2,\ s) \Downarrow (\underline{n_1}\ nop\ \underline{n_2},\ s)} \Downarrow OP_1$$

where P_1 and P_2 should be of same data type which can be either integer or real.

$$CharacterExpression \stackrel{def}{=\!=} \frac{(P_1,\ s) \Downarrow (\underline{char_1},\ s)\ (P_2,\ s) \Downarrow (\underline{char_2},\ s)}{(P_1\ cop\ P_2,\ s) \Downarrow (\underline{char_1}\ cop\ \underline{char_2},\ s)} \Downarrow OP_2$$

$$StringExpression \stackrel{def}{=\!=} \frac{(P_1,\ s) \Downarrow (\underline{str_1},\ s)\ (P_2,\ s) \Downarrow (\underline{str_2},\ s)}{(P_1\ sop\ P_2,\ s) \Downarrow (\underline{str_1}\ sop\ \underline{str_2},\ s)} \Downarrow OP_3$$

$$BooleanExpression \stackrel{def}{=\!=} \frac{(P_1,\ s) \Downarrow (\underline{n_1},\ s)\ (P_2,\ s) \Downarrow (\underline{n_2},\ s)}{(P_1\ bop\ P_2,\ s) \Downarrow (\underline{n_1}\ bop\ \underline{n_2},\ s)} \Downarrow OP_4$$

$$BooleanExpression - NOT \stackrel{def}{=\!=} \frac{(P,\ s) \Downarrow ((\underline{n},\ s),\ s)}{(P,\ s) \Downarrow (\sim n,\ s)} \Downarrow OP_4$$

Rule 4: Empty instruction does nothing with no change of state.

$$\frac{}{(empty,\ s) \Downarrow (empty,\ s)} \Downarrow EMPTY$$

Rule 5: Break Instruction

- stops the execution of the nearest enclosing loop statement or switch statement in which it is present,
- passes the control out of the enclosing loop to the instruction following the loop with change of state.

$$\frac{}{(break,\ s) \Downarrow (break,\ s')} \Downarrow BREAK$$

Rule 6: Stop instruction stops the execution of the program.

$$\frac{}{(stop,\ s) \Downarrow (stop,\ \phi)} \Downarrow STOP$$

Rule 7: Continue instruction passes the control to the end of the loop's body where it is present with a change of state.

$$\frac{}{(continue,\ s) \Downarrow (continue,\ s')} \Downarrow CONTINUE$$

Rule 8: Timer instruction delays the program by a specific number of milliseconds with no change of state.

$$\frac{}{(timer,\ s) \Downarrow (timer,\ s)} \Downarrow TIMER$$

Rule 9: Assignment instruction sets the value of the variable on the left hand side (LHS) of the equal sign to the result of evaluating the expression on the right hand side (RHS). The data types of both the RHS and LHS are the same.

$$\frac{(P,\ s) \Downarrow (\underline{n},\ s)}{(l := P,\ s) \Downarrow (empty,\ s(l \rightarrow n))} \Downarrow ASGNMINT$$

Rule 10: Catenation shows the order of instructions to be executed.
Let P_1 and P_2 are instructions in sequence where P_1 has to be executed first which is to be followed by P_2. The catenation is represented as P_1; P_2.

$$\frac{(P_1,\ s_1) \Downarrow (empty,\ s_2)(P_2,\ s_2) \Downarrow (empty,\ s_3)}{(P_1;\ P_2,\ s_1) \Downarrow (empty,\ s_3)}$$

Rule 11: Read instruction reads the data from the user and assigns it to the respective memory location.

$$\frac{R_1\ R_2 \ldots R_n}{(read(l_1,\ l_2,\ \ldots l_n),\ s) \Downarrow (empty,\ s(l_1 \rightarrow \underline{c_1},\ l_2 \rightarrow \underline{c_2},\ \ldots,\ l_n \rightarrow \underline{c_n}))} \Downarrow READ$$

$$R_1\ is\ (read(l_1),\ (read(l_2,\ \ldots,\ l_n))) \Downarrow (l_1 \rightarrow \underline{c_1},\ s(l_2 \rightarrow \underline{c_2},\ \ldots,\ l_n \rightarrow \underline{c_n}))$$

$$R_2\ is\ (read(l_1),\ read(l_2),\ (read(l_3,\ \ldots,\ l_n))) \Downarrow (l_1 \rightarrow \underline{c_1},\ l_2 \rightarrow \underline{c_2},\ s(l_3 \rightarrow \underline{c_3},\ \ldots,\ l_n \rightarrow \underline{c_n}))$$

$$R_n\ is\ (read(l_1),\ read(l_2),\ ,\ldots,\ read(l_n)) \Downarrow (l_1 \rightarrow \underline{c_1},\ l_2 \rightarrow \underline{c_2},\ \ldots,\ l_n \rightarrow \underline{c_n})$$

Rule 12: Print instruction displays the list of constants or values of the variables or expressions.

$$\frac{PR_1\ PR_2,\ \ldots,\ PR_n}{print(P_1|\underline{c_1},\ P_2|\underline{c_2}, \ldots, P_n|\underline{c_n}) \Downarrow (empty,\ s(\underline{n_1},\ \underline{n_2},\ \ldots,\ \underline{n_n}))}$$

where $P_i|\underline{c_i}$ means program expression or a constant.

$$PR_1\ is\ (print(P_1|c_1), (print(P_2|c_2,\ \ldots,\ P_n|c_n))) \Downarrow (print(\underline{n_1}),\ s(print(P_2|c_2,\ \ldots,\ P_n|c_n)))$$

$$PR_2\ is\ (print(P_1|c_1), print(P_2|c_2), (print(P_3|c_3,\ \ldots,\ P_n|c_n))) \Downarrow (print(\underline{n_1}, \underline{n_2}),\ s(print(P_3|c_3,\ \ldots,\ P_n|c_n)))$$

$$PR_n\ is\ (print(P_1|c_1), print(P_2|c_2),\ \ldots,\ print(P_n|c_n)) \Downarrow (print(\underline{n_1}, \underline{n_2},\ \ldots \underline{n_n}))$$

Rule 13: Branch instruction conditionally executes a statement block depending on the value of the condition part of the expression.

$$\frac{(p,\ s_1) \Downarrow (\underline{true},\ s_1)\ (P_1,\ s_1) \Downarrow (empty,\ s_2)}{(if\ p\ then\ P_1\ else\ P_2,\ s_1) \Downarrow (empty,\ s_2)} \Downarrow COND_1\ and$$
$$\frac{(p,\ s_1) \Downarrow (\underline{false},\ s_1)\ (P_2,\ s_1) \Downarrow (empty,\ s_2)}{(if\ p\ then\ P_1\ else\ P_2,\ s_2) \Downarrow (empty,\ s_2)} \Downarrow COND_2$$

Rule 14: Loop instruction executes a set of instructions more than once based on a condition.

$$\frac{(P\_1,\ s_1) \Downarrow (\underline{true},\ s_1)\ (P_2,\ s_1) \Downarrow (empty,\ s_2)\ (while\ P_1\ do\ P_2,\ s_2) \Downarrow (empty,\ s_3)}{(while\ P_1\ do\ P_2,\ s_1) \Downarrow (empty,\ s_3)} \Downarrow LOOP_1$$
$$\frac{(P,\ s_1) \Downarrow (\underline{false},\ s_1)}{(while\ P_1\ do\ P_2,\ s_1) \Downarrow (empty,\ s_1)} \Downarrow LOOP\_2$$

Also, the behaviour of the finite sequences of commands is unchanged by rearranging the sequence tree. This shows that the associative property holds. Let P_1, P_2, P_3 be three expression statements and s and s' are two states. The associative property can be stated as follows:

$$((P_1;\ P_2);\ P_3,\ s) \Downarrow (empty,\ s') \equiv (P_1;\ (P_2;\ P_3),\ s) \Downarrow (empty,\ s')$$

LIPS uses built-in functions which are mapped to the java built-in functions and require no separate specifications.

4.2 Abstract Machine for the Computational Part of LIPS

This section presents the formal description of the computational part of the LIPS Abstract Machine (LAM). The goal of developing LAM is to provide a precise and well defined semantic framework which can be used for the refining and verification of the LIPS compiler. The LAM executes instructions using single step re-write rules. A re-write rule breaks the steps involved in transforming a given expression P into a final value $V(P \Downarrow^e V)$ which is denoted as follows:

$$P \mapsto P_1 \mapsto P_2 \mapsto \ldots V$$

The mechanical process of producing the final state, V, from the given state, P, is called mechanically reproduce.

The LAM comprises of a set of re-write rules to transform the LAM configurations. The

LAM configuration is a triplet denoted as (C, S, s) where

- C is the code to be executed
- S is the stack which can contain a list of integers, real numbers, Boolean, characters, and strings
- s is a state which is the same as that defined in the LIPS operational semantics.

Code C of the LAM configuration:
The code C of the LAM configuration is a list which is formed by the following grammar: Let `ins` specify the set of LAM instructions, `op` be any operator allowed in a LIPS program, l be any location, and $\underline{c}$ be any constant. The symbol '-' is used to denote an empty code where the empty code is specified as

$$\texttt{C} : -$$

The instruction `ins` and code C are defined as follows:

$$\texttt{ins} ::= PUSH(\underline{c}) \mid FETCH(l) \mid OP(op) \mid READ(l_1, l_2, ..., l_n) \mid$$
$$PRINT(l_1 \mid \underline{c_1}, l_2 \mid \underline{c_2}, ..., l_n \mid \underline{c_n}) \mid EMPTY \mid BREAK \mid STOP \mid CONTINUE \mid$$
$$ASGNMNT(l) \mid BR(P_1, P_2) \mid WHILE(P_1 \; do \; P_2) \; C ::= \mid ins : C$$

Stack S of the LAM configuration:
The stack S is produced using the following grammar:
Let $\underline{c}$ be any integer, real, Boolean, string, or character. Let the symbol '-' be used to denote an empty stack which is denoted as $S : -$. Generally, the stack is defined as

$$S ::= - \mid \underline{c} : S$$

State s of the the LAM configuration:
The state s of LAM configuration is the state of the LIPS program at any point of executing an expression.

With the definitions above, the LAM re-write rule can be written using the triplets as follows:

$$(C_1, S_1, s_1) \mapsto (C_2, S_2, s_2)$$

where $\mapsto$ is a binary relation which changes the configuration (C_1, S_1, s_1) to (C_2, S_2, s_2), C_1 and C_2 are the initial and final codes, S_1 and S_2 are the initial and final status of the stacks, and s_1 and s_2 are the initial and final states. The binary relation is defined inductively on the set of all the LAM configurations by a set of rules. Each rule has the following form:

$$\frac{}{(C_1, S_1, s_1) \mapsto (C_2, S_2, s_2)} \; :: \texttt{RULE}$$

The rules do not have hypotheses/premises. The LAM re-writes have the following restricted structure.

Given an expression, the first step is to load the machine to an initial state. The last re-write step must be an instance of a rule in the LAM which denote the successful termination of the machine and produce a final value. An expression evaluates to state 's' with respect to the LAM if there is a series of re-writing rules satisfying the above restriction. The general form of LAM re-write rule, RULE, is as shown below:

$$\boxed{C_1}\boxed{S_1}\boxed{s_1} \mapsto \boxed{C_2}\boxed{S_2}\boxed{s_2}$$

The re-write rules for the LAM are summarised below:

1. **Push** a constant $\underline{n}$ of a specific data type σ into the stack S. σ can take up one of the following data types:

   $$\{\text{int}, \text{real}, \text{bool}, \text{string}, \text{char}\}$$

   The re-write rule may be written as follows:

   $$\boxed{\underline{n} : C}\boxed{\text{S}}\boxed{\text{s}} \mapsto \boxed{\text{C}}\boxed{\underline{n} : \sigma : S}\boxed{\text{s}}$$

   Example 4. The following example shows the **PUSH** operation:

   $$\boxed{\underline{10} : C}\boxed{\text{S}}\boxed{\text{s}} \mapsto \boxed{\text{C}}\boxed{\underline{10} : int : S}\boxed{\text{s}}$$

2. **FETCH** a value from a memory location l and place in the stack S.

   $$\boxed{l : C}\boxed{\text{S}}\boxed{\text{s}} \mapsto \boxed{\text{C}}\boxed{\underline{s(l)} : S}\boxed{\text{s}}$$

   Example 5. Let l be a location which has a value 100 stored in it ($\underline{s(l)} = 100$). Fetching that value and placing it in the stack is specified as follows:

   $$\boxed{l : C}\boxed{\text{S}}\boxed{\text{s}} \mapsto \boxed{\text{C}}\boxed{\underline{100} : S}\boxed{\text{s}}$$

3. Let *op* be a LIPS operator which operates on two operands/program expressions P_1 and P_2 which are of the same data type. The re-write rule for $P_1 \mathbf{op} P_2$ can be stated as below:

   $$\boxed{P_1 \mathbf{op} P_2 : C}\boxed{\text{S}}\boxed{\text{s}} \mapsto \boxed{P_1 : P_2 : \mathbf{op} : \text{C}}\boxed{\text{S}}\boxed{\text{s}}$$

   Example 6. Let P_1 be $\underline{10.6 + 12}$, P_2 be $\underline{5.6 * 2}$ and $+$ be *op*. The re-write rule is

written as as below:

$$\|\underline{10.6+12}+\underline{5.6*2}:C\|\,S\,\|\,s\,\| \mapsto \|\underline{10.6+12}:\underline{5.6*2}:+:C\|\,S\,\|\,s\,\|$$

4. **Assignment instruction** is used to assign the evaluated value of the program expression P to a memory location l, i.e., $l := P$.

$$\|l := P : C\|\,S\,\|\,s\,\| \mapsto \|P : ASGNMNT(l) :\text{C}\|\,S\,\|\,s\,\|$$

$$\|ASGNMNT(l) : C\|\,\underline{c} : S\,\|\,s\,\| \mapsto \|P : ASGNMNT(l) :\text{C}\|\,S\,\|\,s\{l \mapsto \underline{c}\}\|$$

Example 7. Let $l = \underline{10.9}$ be an assignment statement to be executed. The re-write rule can be specified as follows:

$$\|l := \underline{10.9} : C\|\,S\,\|\,s\,\| \mapsto \|\underline{10.9} : ASGNMNT(l) : C\|\,S\,\|\,s\,\|$$

$$\|ASGNMNT(l) : C\|\,\underline{10.9} : S\,\|\,s\,\| \mapsto \|\text{C}\|\,S\,\|\,s\{l \mapsto \underline{10.9}\}\|$$

5. **Empty instruction**

$$\|empty :\text{C}\|\,S\,\|\,s\,\| \mapsto \|\text{C}\|\,S\,\|\,s\,\|$$

6. **Break instruction**

$$\|Break :\text{C}\|\,S\,\|\,s\,\| \mapsto \|\text{C}\|\,S\,\|\,s\,\|$$

7. **Stop instruction**

$$\|Stop :\text{C}\|\,S\,\|\,s\,\| \mapsto \|\text{C}\|\,S\,\|\,-\,\|$$

8. **Continue instruction**

$$\|continue :\text{C}\|\,S\,\|\,s\,\| \mapsto \|\text{C}\|\,S\,\|\,s\,\|$$

9. **READ** instruction

$$\|read(l_1, l_2, ..., l_n) :\text{C}\|\,S\,\|\,s\,\| \mapsto \|\text{C}\|\,S\,\|\,s\{l_1 \mapsto \underline{c_1}, l_2 \mapsto \underline{c_2}, ..., l_n \mapsto \underline{c_n}\}\|$$

Example 8. Let **read** reads a list of finite number of constants $\underline{c_1}, \underline{c_2}, ..., \underline{c_n}$ and store them in memory location $l_1, l_2, ..., l_n$. The re-write is given below:

$$\|read(l_1, l_2) : C\|\,S\,\|\,s\,\| \mapsto \|\text{C}\|\,l_1 \leftarrow \underline{16}:\ l_2 \leftarrow \underline{25} : read :\ \text{S}\|\,s\,\|$$

$$\| \text{C} \| l_1 \leftarrow \underline{16}:\ l_2 \leftarrow \underline{25} : read:\ \text{S} \| \text{s} \| \mapsto \| \text{C} \| \text{S} \| s\{l_1 \leftarrow \underline{16}:\ l_2 \leftarrow \underline{25}\} \|$$

10. **print** instruction: Let $\underline{c_1}$, $\underline{c_2}$, ...,$\underline{c_n}$ be constant values, $P_1, P_2, ..., P_n$ be a set of program expressions and $\underline{n_1}$, $\underline{n_2}$, ...,$\underline{n_5}$ be the set of evaluated values for the program expressions respectively to be printed. The re-write for the print instruction is given below:

$$\| print(c_1|P_1 := \underline{n_1} : c_2|P_2 := \underline{n_2} : ... : c_n|P_n := \underline{n_n}) : \text{C} \| \text{S} \| \text{s} \| \mapsto \| \text{C} \| \underline{n_1}, \underline{n_2}, ..., \underline{n_n} : \text{S} \| \text{s} \|$$

$$\| print(\underline{n_1}, \underline{n_2}, ..., \underline{n_n}) : C \| \text{S} \| \text{s} \| \mapsto \| \text{C} \| \underline{n_1}, \underline{n_2}, ..., \underline{n_n} : S \| \text{s} \|$$

$$\| \text{C} \| \underline{n_1}, \underline{n_2}, ..., \underline{n_n} : S \| \text{s} \| \mapsto \| \text{C} \| \text{S} \| s\{\underline{n_1}, \underline{n_2}, ..., \underline{n_n}\} \|$$

Example 9. A re-write rule to print integers 16 and 25

$$\| print(\underline{16}, \underline{25} : C \| \text{S} \| \text{s} \| \mapsto \| \text{C} \| \underline{16}, \underline{25} : S \| \text{s} \| \mapsto \| \text{C} \| \text{S} \| \text{s}\{\underline{16},\underline{25}\} \|$$

11. **Catenation instruction** Let P_1 and P_2 be the two program expressions to be executed in sequence. The re-write rule is given below:

$$\| (P_1; P_2) : C \| \text{S} \| \text{s} \| \mapsto \| P_1 : P_2 : C \| \text{S} \| \text{s} \|$$

12. **Branch instruction** Let P_1 be a program expression to be executed if the condition is true ($\underline{T}$) and P_2 be the program expression to be executed if the condition is false ($\underline{F}$).

$$\| BR(P_1; P_2) : C \| \underline{T} : S \| \text{s} \| \mapsto \| P_1 : C \| \text{S} \| \text{s} \|$$

$$\| BR(P_1; P_2) : C \| \underline{F} : S \| \text{s} \| \mapsto \| P_2 : C \| \text{S} \| \text{s} \|$$

13. **Loop instruction** Let P_1 and P_2 be two program expressions where P_2 will be executed until P_1 is true. The re-write rule is given below:

$$\| while\ (P_1\ do\ P_2) : C \| \underline{T} : S \| \text{s} \| \mapsto \| BR((P_2; while\ (P_1\ do\ P_2)), EMPTY) : C \| \text{S} \| \text{s} \|$$

4.2.1 Compilation of LIPS Program Expressions into LAM Codes

A location in LAM can store a value which is any one of the allowed data types. The LAM must start in a known state. It is assumed that the program expressions are type checked. This section describes the compilation of the LIPS expressions into the LAM code. Let

$$[\![-]\!] : EXP \rightarrow LAMcodes$$

be the function which takes a LIPS program expression and compiles it to a LAM code. The PUSH and FETCH operations are specified using rules shown in Table 4.7

Table 4.7: Compilation of LIPS Expressions into LAM Code

Instruction	Rules
PUSH	$[\![\underline{c}]\!] \stackrel{def}{=\!=} PUSH(\underline{c})$
FETCH	$[\![l]\!] \stackrel{def}{=\!=} FETCH(l)$
operator op	$[\![P_1 \; op \; P_2]\!] \stackrel{def}{=\!=} [\![P_1]\!] : [\![P_2]\!] : OP(op)$
Assignment	$[\![l := P]\!] \stackrel{def}{=\!=} [\![P]\!] : ASGNMNT$
EMPTY	$[\![empty]\!] \stackrel{def}{=\!=} EMPTY$
BREAK	$[\![break]\!] \stackrel{def}{=\!=} BREAK$
STOP	$[\![stop]\!] \stackrel{def}{=\!=} STOP$
CONTINUE	$[\![continue]\!] \stackrel{def}{=\!=} CONTINUE$
TIMER	$[\![timer]\!] \stackrel{def}{=\!=} TIMER$
READ	$[\![read(l_1, l_2, ..., l_n)l]\!] \stackrel{def}{=\!=} READ(l_1, l_2, ..., l_n)$
PRINT	$[\![print(P_1\|\underline{c_1}, P_2\|\underline{c_2}, ..., P_n\|\underline{c_n})l]\!] \stackrel{def}{=\!=} PRINT(P_1\|\underline{c_1}, P_2\|\underline{c_2}, ..., P_n\|\underline{c_n})$
Branch	$[\![if \; P \; then \; P_1 \; P_2]\!] \stackrel{def}{=\!=} [P] : BR([\![P_1]\!][\![P_2]\!])$
Loop	$[\![while \; P_1 \; do \; P_2]\!] \stackrel{def}{=\!=} WHILE([\![P_1]\!] \; do \; [\![P_2]\!])$

4.3 Operational Semantics for the Communication Part of LIPS

As we are modelling the asynchronous message passing for LIPS, we need to model how different statements are executed for message passing. Structural Operational Semantics (SOS) or small-step semantics can express parallelism by using interleaving steps. Execution of statements using SOS is described by one or more transitions and can capture both the result of computation and how the result is produced. These reasons led us to choose SOS along with the big-step semantics to describe the asynchronous communication in LIPS which is implemented using the Asynchronous Message Passing System (AMPS). Once the semantics has been defined for AMPS and the other statements involved in message passing, it can be used as a technical reference manual.

AMPS consists of a Data Structure (DS), a Data Matrix (DM) and a set of interface codes which are implemented using a set of function calls. These are generated implicitly by the LIPS compiler and therefore it is decided to hide some of the atomic actions performed by these functions. SOS for asynchronous process description languages are usually described by the Labelled Transition Systems (LTS). We illustrate the LTS for the communication part of LIPS by formally defining syntactical categories. From these formal definitions, we demonstrate the LTS.

In order to describe the SOS for the communication part of LIPS, the primitives of AMPS

Table 4.8: Extended Data Types for the Communication Part of LIPS

Name of the Type	Purpose
Channel	Means of communication that carries data belonging to allowed types.
Flag	Binary data type which can be set or reset to show the availability of the data.
node_number	Number of the node from where the data is received or to where the data is sent.
node_name	String data type to show the name of the node.
Vnum	Variable number - A number is assigned to the channel variables in the network.
Vname	Variable Name - A string type data to represent the name of a variable.
counter	Number showing the number of nodes that are to receive the data.
type_number	A unique integer number assigned to each type of data - 1 .. 9.
Data	Data in string form participating in the message passing.
data_packet	A combination of node_number, vnum, type_number, data which is passed between the nodes during message passing.
InList	A singly linked list used to store the list of input channels and related information. It consists of the Vnum, Vname, Flag, Data, and a link to the next input channel in the list.
OutList	A singly linked list used to store the list of output channels and related information. It consists of Vnum, Vname, counter, Data and link to the next output channel in the list.
DS	A data structure which is a singly linked list where all the nodes in the network are linked to all the other nodes in the network. It consists of node_number, node_name, InList, OutList, and a link to the next node.
DM	A two dimensional matrix which contains the details of channel variables in the networking: Vnum, node_number, type_number, status of all the nodes in the network to where the channel could be sent as input.

and the communication schema are described.

4.3.1 Primitives and Communication Schema for the Asynchronous Message Passing in the LIPS

The AMPS of LIPS makes three main function calls and the data involved in the message passing is always sent to the Data Structure (DS) or received from the DS thereby the sender or receiver never waits for the recipient or the sender respectively. The basic types of data have been extended with the additional data types and functions to handle the asynchronous communication. Table 4.8 shows the extended data types needed to implement the message passing. Table 4.9 lists the functions used in the AMPS of LIPS.

Table 4.9: Functions Used in the AMPS of LIPS

Name of the Function	Purpose
IS_ok_to_send (Src_node_number, Vnum)	Sender checks whether it can send data to the AMPS.
Is_input_available(node_number, Vnum)	Receiver checks the availability of the data in the AMPS.
Send(data_packet)	Sender sends the data packet.

When one of these functions is called, the set of statements which are executed belong to the computational part of LIPS and big-step semantics can be used to define the operational behaviour.

4.3.2 Communication Schema for Asynchronous Communication

The communication conventions essential for asynchronous communication are described by the following:

- Guarded processes which make up the node.
- Node with its input and output channels.
- Connect statements needed to express the topology of the network.
- Functions used by the AMPS of LIPS to perform message passing.

A guarded process in LIPS consists of a guard and a statement block. The guard is a collection of channels which have to hold valid data for the statement block to be executed. Let GP be the set of guarded processes in a process node where,

$$\mathtt{GP} = \{gp_1,\ gp_2,\ gp_3,\ \ldots,\ gp_n\}$$

Let G_1, G_2, G_3, ..., Gn be the guards corresponding to the guarded processes respectively.

$$\forall i\ :\ 1 \leq i \leq n\ :\ ch_{i1},\ ch_{i2},\ ch_{i3},\ \ldots,\ ch_{im}$$

are the data channels and m varies between 0 and l.

Let fch_{i1}, fch_{i2}, fch_{i3}, ..., fch_{im} be the flags associated with the data channels where fch_{ij} will be set to true if data is available in ch_{ij} where i ranges between 1 and n and j ranges between 0 and l.

The code to execute the set of guarded processes in a node can be given as follows:

```
while (true) do {
    if (G1) {statement_block_1}
    else
    if (G2) {statement_block_2}
    else
    .
    .
    .
    else
    if (Gn) {statement_block_n}
}
```

where
- n, number of guards, is an integer.
- $\forall i : 1 \leq i \leq n$, *statement_block_i* for G_i will be executed only when the data is available in all of the channels. If fch_{i1}, fch_{i2}, fch_{i3}, ..., fch_{im} are the flags to the input channels for G_i, then $fch_{i1} \wedge fch_{i2} \wedge fch_{i3} \wedge ..., \wedge fch_{im}$ should be true to confirm the availability of data in the input channels.

The following are the communication schema:

1. **guard G_i:**

   $$G_i = fch_{i1} \wedge fch_{i2} \wedge fch_{i3} \wedge ..., \wedge fch_{im}$$

   If G_i is true then new values are available on the corresponding data channels ch_{i1}, ch_{i2}, ch_{i3}, ..., ch_{im}.

2. **if-statement/alternate construct for a guarded process GP_i:**

   $$if\ (fch_{i1} \wedge fch_{i2} \wedge fch_{i3} \wedge ... \wedge fch_{im})\ then\ P_{i1};\ P_{i2};\ P_{i3};\ ...,\ P_{ik}$$

   where $\forall j : 0 \leq j \leq m$, fch_{ij} are flags to channels which will be set to true when their respective channels have valid data and P_{i1}; P_{i2}; P_{i3}; ..., P_{ik} be the sequence of program expressions. If the guard becomes true then the associated sequence of program expressions will be executed and values for 0 or more output channels will be generated.

3. **while statement for a node consisting of a set of guarded processes GP:**

   Let n be the number of guarded processes in a node and let m be the number of input channels in each guarded process where m varies between 0 and l. The node can be constructed with a set of alternate constructs embedded by a while statement as

follows:

$$
\left\{
\begin{array}{l}
\textit{while } (\textit{true}) \textit{ do} \\
\quad \textit{if } (fch_{11} \wedge fch_{12} \wedge fch_{13} \wedge ... \wedge fch_{1m}) \\
\quad\quad \textit{then } P_{11}; P_{12}; P_{13}; ..., P_{1k} \\
\quad \textit{else} \\
\quad \textit{if } (fch_{21} \wedge fch_{22} \wedge fch_{23} \wedge ... \wedge fch_{2m}) \\
\quad\quad \textit{then } P_{21}; P_{22}; P_{23}; ..., P_{2k} \\
\quad \textit{else} \\
\quad \vdots \\
\quad \textit{else} \\
\quad \textit{if } (fch_{n1} \wedge fch_{n2} \wedge fch_{n3} \wedge ... \wedge fch_{nm}) \\
\quad\quad \textit{then } P_{n1}; P_{n2}; P_{n3}; ..., P_{nk}
\end{array}
\right.
$$

4. **Connect Statement:**

   Let $R = R_1, R_2, ..., R_n$ be the set of of nodes in a system under consideration where $n \geq 1$,

   Let $ch = ch_1, ch_2, ..., ch_k$ be the set of channels/ports to be used in the system where $k \geq 0$,

   Let $ich_{11}, ich_{12}, ..., ich_{1m} \in ch$ be a set of input channels for a node, say R_1 where $m \geq 0$, and

   Let $och_{11}, och_{12}, ..., och_{1s} \in ch$ be a set of output channels for a node, say R_1 where $s \geq 0$.

   The connect statement for the node R_1 is written as follows:

$$R_1(ich_{11} \wedge ich_{12} \wedge ... \wedge ich_{1m}) \rightarrow (och_{11} \wedge och_{12} \wedge ... \wedge och_{1s})$$

5. **The function to check whether input data is available or not:**

   The result of this function call is either 1 or a 0 indicating the availability of data. If a 1 is received, then the guard will ask for the data to be sent to it. The function is specified as:

Is_input_available(node_number, vnum)

6. **The function to find whether data can be sent or not:**
   This returns a value 1 or 0. If 1 is received, then the second function is initiated which sends the data as a data packet. If 0 is received, the sender waits until it gets a 1.

Is_ok_to_send(Src_node_number, vnum)

7. **The function to send the data:**

Send(data_packet)

The same send function is used by the data structure to send the data packet to a requested node.

The seven schema, based on the working of AMPS, form the major communication rules used in the LIPS language.

4.3.3 Syntactic Categories for Asynchronous Communication

The existing data types have to be extended and the extended data types will be used implicitly by the LIPS compiler. The extended data types are given as below:

σ ::= `int` | `real` | `bool` | `string` | `char` | `channel`
| `flag` | `node_number` | `node_name` | `counter` | `vnum` | `vname`
| `type_number` | `data_packet` | `inlist` | `outlist` | `data_structure` | `cmd`

According to the extended data types, the syntactic categories of LIPS have been extended and are listed below:

i. **Set of Channel Numbers - positive integer values:**

$$\texttt{CHANNEL} \stackrel{def}{=\!=} \{ch_1, ch_2, ..., ch_n\}$$

ii. **Set of flags which can take Boolean values:**

$$\texttt{FLAG} \stackrel{def}{=\!=} \{fch_1, fch_2, ..., fch_n\}$$

iii. **Set of node numbers**

$$\texttt{NODE\_NUMBER} \stackrel{def}{=\!=} \text{k } \{\text{finite set of integers}\}$$

iv. **Set of node names**

$$\texttt{NODE\_NAME} \stackrel{def}{=\!=} \{\text{finite set of integers}\}$$

v. **Set of Channel Variable Numbers**

$$\texttt{VNUM} \stackrel{def}{=\!=} \{\text{finite set of integers}\}$$

vi. **Data Type Numbers**

TYPE_NUMBER $\stackrel{def}{=\!=}$ {1, 2, 3, 4, 5, 6, 7, 8, 9} Type number is the number assigned to each type of data. For example, 1 is assigned to integer data.

vii. **The original data in string form**

$$\texttt{DATA} \stackrel{def}{=\!=} \{\text{data} \mid \text{data} \in \textbf{STR}\}$$

viii. **Data Packet**

$$\texttt{DATA\_PACKET} \stackrel{def}{=\!=} \text{<node\_number, vnum, type\_number, data>}$$

ix. **List of Input channels**

InList $\stackrel{def}{=\!=}$

```
Struct inlist {
        vnum, vname, flag, data,
        Struct inlist *next}
```

x. **List of Output channels**

OutList $\stackrel{def}{=\!=}$

```
Struct outlist {
     vnum, vname, counter, data,
     Struct outlist *next}
```

xi. **Data Structure**

DS $\stackrel{def}{=\!=}$

```
Struct data_struct {
        nnum, nname, inlist, outlist,
        Struct data_struct *next}
```

xii. **Data Matrix**

$$\texttt{Data\_Matrix} \stackrel{def}{=\!=} \text{DM [m][n]}$$

where m is the number of rows which is equal to the number of variables and n is the number of columns which is equal to the number of nodes in the network plus 3. The various columns in the network are Vnum, node_number, type_number, node_number of the nodes in the network.

xiii. **Checking the availability of data**

$$\texttt{IsInputAvail} \stackrel{def}{=\!=} \text{Is\_input\_available(node\_number, vnum)}$$

When this function is called, a set of actions take place which are represented by the following code segment:

```
int availstatus = 0
while (DS != null) do {
    if(DS.node_number = node_number) then{
        while(inlist != null) do {
            if (inlist.vnum = vnum) then{
                if (inlist.flag = 1) then{
                    availstatus = 1
                    break whileloop
                }
            }
            inlist = inlist.next
        }
          if (availstatus = 1) then
                    break whileloop
      }
      DS = DS.next
      }
return availstatus
```

xiv. **Checking the possibility of sending the data**

$$\texttt{ISOKTOSEND} \stackrel{def}{=\!=} \text{Is\_ok\_to\_send(Src\_node\_number, vnum)}$$

When this function is called, a set of actions take place which are represented by the following code segment:

```
int status = 0
while (DS != null){
    if(DS.node_number = Src_node_number) then {
        while(outlist != null) do {
            if (outlist.vnum = vnum) then {
                if (outlist.counter = 0) then{
                    status = 1
```

```
                    break whileloop
                }
            }
            outlist = outlist.next
        }
        if (status = 1) then
                    break whileloop
    }
    DS = DS.next
    }
return status
```

xv. **Sending data**

$$\text{SEND} \stackrel{def}{=\!=} \text{Send(data\_packet)}$$

where the data_packet consists of the node number, variable number, vnum, type of data, type_number, and the actual data, data, in string form. On the receipt of this packet, the AMPS checks the data structure, DS, to see whether the variable number, and type are correct and stores the data in the appropriate field. The counter is set to the number of nodes that are to receive the data by consulting the data matrix, DM, which consists of m number of rows and n number of columns where m is the number of variables where n is (3 (to store the variable number, its source node number and type number) + number of nodes in the network). The Send function returns a 1 to indicate a success. For a given node number, variable number and its type, the code segment to find the value for the counter by using the data matrix is given below:
Let i, j be two integers and let i=0 and j=3.

```
int counter = 0
while (i <= m -1) do {
      if (DM[i][0] = vnum) then{
        if ((DM[i][1] = node_number) ^ (DM[i][2] = type_number)) then{
            while (j <= n - 1) do{
               if (DM[i][j] = 1) then
                counter = counter + 1
            }
        }
      }
}
return counter
```

The code segment to place the data in the data structure and set the counter value is give below:

```
int status = 0
while (DS != null) do {
    if(DS.node_number = Src_node_number) then{
        while(outlist != null) do{
            if (outlist.vnum = vnum) then{
                outlist.data = data
                outlist.counter = counter
                status = 1
            }
        }
    }
}
return status
```

After storing the data, the AMPS consults the DM, distributes the data to other DS nodes, and decrements the copy counter accordingly. Here the data is written to the input channel variable of a receiving DS node, provided the status counter of that input channel variable is 0 (that is, the channel is free to receive new data). Once the data is received, the status is set to 1. If any of the DS destination nodes are unable to receive the new data, the AMPS periodically checks whether they are free to accept the data.

```
while (DS != null){
    if(DS.node_number = node_number) then{
        while(inlist != null) do{
            if ((inlist.vnum = vnum) ^ (inlist.flag = 0)) then{
                inlist.data = data
                inlist.flag = 1
                counter = counter - 1
            }
        inlist = inlist.next
 }}
}
return counter;
```

xvi. **Guard**

$$G_i \stackrel{def}{=\!=} fch_{i1} \wedge fch_{i2} \wedge fch_{i3} \wedge ..., \wedge fch_{im}$$

xvii. **Guarded Process**

$$gp_i \stackrel{def}{=\!=} \text{if } (fch_{i1} \wedge fch_{i2} \wedge fch_{i3} \wedge ... \wedge fch_{im}) \text{ then } P_{i1}; P_{i2}; P_{i3}; ..., P_{ik}$$

xviii. **Node**

$$\begin{cases} R_j \stackrel{def}{=\!=} \\ while\ (true)\ do \\ \qquad if\ \ (fch_{i1} \wedge fch_{i2} \wedge fch_{i3} \wedge ... \wedge fch_{im})\ then\ \ P_{i1}; P_{i2}; P_{i3}; ..., P_{ik} \\ od \end{cases}$$

for all $1 \leq i \leq n$ where n is the number of guarded processes for a node $R_j \in R$.

xix. **Connect Statement**

$$\begin{cases} \texttt{CONNECT} \stackrel{def}{=\!=} \\ \qquad R_i(ich_{i1} \wedge ich_{i2} \wedge ... \wedge ich_{im}) \rightarrow (och_{i1} \wedge och_{i2} \wedge ... \wedge och_{is}) \end{cases}$$

where $R_i \in R$
$ich_{i1}, ich_{i2}, ..., ich_{im} \in ch$
$och_{i1}, och_{i2}, ..., och_{is} \in ch$

The type assignments of LIPS have been extended and are given in Table 4.10.

Table 4.10: Extended Type Assignments P :: σ of LIPS

$\overline{\underline{ch} :: \texttt{channel}}\ [\forall ch \in \mathbf{CH}] :: \texttt{CHANNEL}$	$\overline{l : \texttt{channel}}\ l :: \mathbf{CH} \in \texttt{L} :: \texttt{CHANNELLOC}$
$\overline{\underline{fch} :: \texttt{flag}}\ [\forall fch \in \texttt{BOOLEAN}] :: \texttt{FLAG}$	$\overline{l : \texttt{flag}}\ l :: \texttt{flag} \in \texttt{L} :: \texttt{FLAGLOC}$
$\overline{\underline{n} :: \texttt{nodenum}}\ [\forall n \in \mathbf{Z}] :: \texttt{NODE\_NUMBER}$	$\overline{l : \texttt{nodenum}}\ l :: \texttt{nodenum} \in \texttt{L} :: \texttt{INTLOC}$
$\overline{\underline{n} :: \texttt{vnum}}\ [\forall n \in \mathbf{Z}] :: \texttt{VNUM}$	$\overline{l : \texttt{vnum}}\ l :: \texttt{vnum} \in \texttt{L} :: \texttt{INTLOC}$
$\overline{\underline{n} :: \texttt{typenum}}\ [\forall n \in \mathbf{Z}] :: \texttt{TYPE\_NUMBER}$	$\overline{l : \texttt{typenum}}\ l :: \texttt{typenum} \in \texttt{L} :: \texttt{INTLOC}$

$\frac{}{str :: \texttt{data}}[\forall str \in \mathbf{STR}] :: \texttt{DATA}$	$\frac{}{l : \texttt{data}}\, l :: \texttt{data} \in \texttt{L} :: \texttt{STRINGLOC}$
$\frac{}{n :: \texttt{counter}}[\forall n \in \mathbf{Z}] :: \texttt{COUNTER}$	$\frac{}{l : \texttt{counter}}\, l :: \texttt{counter} \in \texttt{L} :: \texttt{INTLOC}$
$\frac{}{str :: \texttt{vname}}[\forall str \in \mathbf{STR}] :: \texttt{VNAME}$	$\frac{}{l : \texttt{vname}}\, l :: \texttt{vname} \in \texttt{L} :: \texttt{STRINGLOC}$
$\frac{}{str :: \texttt{node\_name}}[\forall str \in \mathbf{STR}] :: \texttt{NODE\_NAME}$	$\frac{}{l : \texttt{node\_name}}\, l :: \texttt{node\_name} \in \texttt{L} :: \texttt{STRINGLOC}$
$\frac{}{array[][] :: \texttt{DM[m][n]}}[\forall m, n \in \mathbf{Z}] :: \texttt{DM[m][n]}$	
$\frac{D_1\ D_2\ D_3\ D_4\ D_5}{InList \stackrel{def}{=\!=} \{Struct\ vnum, vname, flag, data, Struct\ inlist * next\}} :: \texttt{INLIST}$ $D_1\ is\ vnum :: \texttt{VNUM}\ D_2\ is\ vname :: \texttt{VNAME}$ $D_3\ is\ flag :: \texttt{FLAG}\ D_4\ is\ data :: \texttt{DATA}\ D_5\ is\ struct\ inlist * next :: InList$	
$\frac{D_1\ D_2\ D_3\ D_4\ D_5}{OutList \stackrel{def}{=\!=} \{Struct\ vnum, vname, flag, data, Struct\ outlist * next\}} :: \texttt{INLIST}$ $D_1\ is\ vnum :: \texttt{VNUM}\ D_2\ is\ vname :: \texttt{VNAME}$ $D_3\ is\ flag :: \texttt{FLAG}\ D_4\ is\ data :: \texttt{DATA}\ D_5\ is\ struct\ outlist * next :: OutList$	
$\frac{D_1\ D_2\ D_3\ D_4\ D_5}{Data\_Struc \stackrel{def}{=\!=} \{Struct\ nnum, node\_name, inlist, outlist, Struct\ Data\_Struc * next\}} :: \texttt{DS}$ $D_1\ is\ nnum :: \texttt{NNUM}\ D_2\ is\ node\_name :: \texttt{NODE\_NAME}$ $D_3\ is\ inlist :: \texttt{INLIST}\ D_4\ is\ outlist :: \texttt{OUTLIST}$ $D_5\ is\ struct\ Data\_Struct * next :: DS$	

$$\frac{D_1\ D_2\ D_3\ D_4}{DATA\_PACKET \stackrel{def}{=\!=} \{node\_number, vnum, type\_number, data\}} :: \texttt{DATA\_PACKET}$$

D_1 *is node_number* :: NODE_NUMBER D_2 *is vnum* :: VNUM

D_3 *is type_number* :: TYPE_NUMBER D_4 *is data* :: DATA

$$\frac{node\_number :: \texttt{NODE\_NUMBER}\ vnum :: \texttt{VNUM}}{Is\_input\_available(node\_number,\ vnum) :: \texttt{cmd}} :: \texttt{IS\_INPUT\_AVAILABLE}$$

$$\frac{src\_node\_number :: \texttt{NODE\_NUMBER}\ vnum :: \texttt{VNUM}}{Is\_ok\_to\_send(Src\_node\_number,\ vnum) :: \texttt{cmd}} :: \texttt{IS\_OK\_TO\_SEND}$$

$$\frac{data\_packet, :: \texttt{DATA\_PACKET}}{Send(data\_packet) :: \texttt{cmd}} :: \texttt{SEND}$$

There are no separate type assignment statements needed for the node and guarded process as they make use of the existing while and alternate construct.

4.3.4 Structural Operational Semantics (SOS) for the Asynchronous Communication

In SOS, the behaviour of the processes is modelled using the Labelled Transition System (LTS). These transitions are caused by the inference rules that follow the syntactic structure of the processes.

Definition 1. Labelled Transition System
A Labelled Transition System (LTS) is a triplet {S, K, T} where:

- S is a set of states,
- K is a set of labels where $\overline{K}$= $\{\overline{k} |\ k \in K\}$,
- T = $\{\xrightarrow{k}, k \in K\}$ is a transition relation where $\xrightarrow{k}$ is a binary transition relation on S.

The transition can be written as $s \xrightarrow{k} s'$ instead of (s,s')$\in\xrightarrow{k}$.

LTS is a set of inference rules used to specify the operational semantics of the calculus. It is defined using the syntactical structure of the term defining the processes and it

describes the observational semantics. The general form of SOS for a function can be defined as follows:

Let f be the name of the function.
Let $x = \{x_1, x_2, ... xn\}$ be the set of argument parameters associated with the function.
Let $x_i : 1 \leq i \leq n$ where

type of $x_i \in$ CHANNEL $\vee$ FLAG $\vee$ NODE_NUMBER $\vee$ VNUM $\vee$ VNAME $\vee$ TYPE_NUMBER $\vee$ DATA $\vee$ DATA_PACKET $\vee$ COUNTER $\vee$ DATA_STRUCTURE $\vee$ INLIST $\vee$ OUTLIST.

Let y be the value returned by the function where y is either a 0 or a 1.
Let s_1 and s_2 be the initial and final state of the caller respectively.
The SOS is

$$\frac{}{(f(x_1, x_2, ... x_n), s_1) \xrightarrow{x_1, x_2, ..., x_n} (y, s_2)} :: f$$

Following are the inference rules used to specify the SOS for the asynchronous message passing in LIPS.

1. **Guard and Guarded Process**

   The SOS for a guarded process GP_i is given below:

   Let $FCH_i = \{fch_{i1}, fch_{i2}, ..., fch_{im}\}$ be the set of flags associated with the input channels $CH_i = \{ch_{i1}, ch_{i2}, ..., ch_{im}\}$ respectively for some positive integer $m \geq 0$. Let $G_i = (fch_{i1}, \wedge fch_{i2}, \wedge ... \wedge fch_{im})$ be a guard or condition to be satisfied for the associated process body to be executed.

   Let GN_i is guard number and VNUM is variable number. For a fch_{ij} to be true, $Is\_input\_available(GN_i, VNUM\ of\ ch_i)$ should return 1. When a guard in a node requires an input, it makes the $Is\_input\_available$ function call to the AMPS. When a 1 is returned, the node automatically initiate the Send function which sends the data from the Data Structure (DS) of the AMPS.

   The SOS for a guard G_i is specified as follows:

   $$\frac{(fch_{i1} := \underline{T}, s_1) \xrightarrow{\underline{T}} (ch_{i1}, s) \quad \cdots \quad (fch_{im} := \underline{T}, s_1) \xrightarrow{\underline{T}} (ch_{im}, s)}{(fch_{i1} \wedge fch_{i2} \wedge ... \wedge fch_{im}, s_1) \xrightarrow{\underline{T}} (\underline{T}\{ch_{i1}, ch_{i2}, ..., ch_{im}\}, s_2)} :: G_i$$

   Let $P_i = P_{i1}; P_{i2}; ...; P_{ik}$ be the set of statements in the process body for some $k \geq 0$. These statements may contain assignment statements to assign values for the output channels. When such a statement is encountered, the function, $Is\_ok\_to\_send$ will be called. If this call returns a 1 then the Send function will be called to send the data to the DS of the AMPS.

   Let $OCH_i = \{val_{i1}, val_{i2}, ..., val_{is}\}$ be the set of values associated with the output

channels for the guard G_i where $s \geq 0$.

The SOS for the guarded process GP_i is specified as follows:

$$\frac{\overline{(G_i, s_1) \xrightarrow{\underline{T}} (\underline{T}, s_1\{G_i = \underline{T}\})} \quad \overline{(P_i, s_1) \xrightarrow{CH_i} (OCH_i, s_2\{G_i = \underline{F}\})}}{(if\ G_i\ then\ P_i,\ s_1) \xrightarrow{CH_i} (OCH_i,\ s_2\{G_i = \underline{F}\})} :: GP_i$$

2. **Node**

The node $R_i \in R$, which is a collection of guarded processes, is illustrated using an infinite while loop. The SOS for a node using `while statement` is given below:

Let $GP = \{GP_1, GP_2, ..., GP_n\}$ be the set of guarded processes where $n \geq 0$.

Let $FCH_i = \{fch_{i1}, fch_{i2}, ..., fch_{im}\}$ be the set of flags associated with the input channels $CH_i = \{ch_{i1}, ch_{i2}, ..., ch_{im}\}$ respectively for some positive integer $m \geq 0$.

Let $OCH_i = \{val_{i1}, val_{i2}, ..., val_{is}\}$ be the set of output channels associated with the guarded process GP_i where $s \geq 0$.

$$\frac{A \quad B}{(while\ (\underline{T})\ do\ (GP_1\ else\ GP_2\ else\ ...\ GP_n),\ s_1) \xrightarrow{\underline{T}} (if\ (\underline{T}\ then\ (GP_i;\ while\ \underline{T}\ do\ \ (GP_1\ else\ GP_2\ else\ ...\ GP_n))),\ s_k)} :: R_i$$

$$A \stackrel{def}{=} \frac{\overline{(G_i, s_j) \xrightarrow{CH_i} (\underline{T}, s_j)} \quad \overline{(P_i, s_j) \xrightarrow{CH_i} (OCH_i, s_j)}}{(\underline{T},\ s_j) \xrightarrow{\underline{T}} (\underline{T},\ s_j)\ \ (if\ G_i\ then\ P_i,\ s_j) \xrightarrow{CH_i} (OCH_i,\ s_{j+1})} :: GP_i$$

where $1 \leq j \leq k$ and k is some positive integer.

$$B \stackrel{def}{=} \frac{A \quad B}{(while\ (\underline{T})\ do\ (GP_1\ else\ GP_2\ else\ ...\ GP_n),\ s_j) \xrightarrow{\underline{T}} (if\ (\underline{T}\ then\ (GP_i;\ while\ \underline{T}\ do\ \ (GP_1\ else\ GP_2\ else\ ...\ GP_n))),\ s_k)} :: R_i$$

3. **Network definition**

A `connect` statement is closely associated with the the node's definition and it specifies the set of input and output channels associated with a node. Details of the connect statement can be found in Section 3.1 of Chapter 3.

Let R_i be a node in a system under consideration.

Let $ICH_i = \{ich_{i1}, ich_{i2}, ..., ich_{im}\}$ be the set of input channels associated with R_i where $m \geq 0$.

Let $OCH_i = \{och_{i1}, och_{i2}, ..., och_{is}\}$ be the set of output channels associated with R_i where $s \geq 0$.

The SOS for one connect statement is given below:

$$\frac{}{(R_i(ich_{i1} \wedge ich_{i2} \wedge ... \wedge ich_{im}),\ s_1) \longrightarrow ((och_{i1}, och_{i2}, ..., och_{is}),\ s_2)} :: Connect$$

A network may consist of more than one node. Let n be the number of nodes in a network and m and s are the number of input channels and output channels

respectively whose value changes for every node in the network. The SOS for the network defined using these n number of connect statements is given below:

$$\frac{}{\forall i : 1 \leq i \leq n((R_i(ich_{i1} \wedge ich_{i2} \wedge ... \wedge ich_{im}),\ s_1) \longrightarrow ((och_{i1}, och_{i2}, ..., och_{is}),\ s_2))} :: Connect$$

4. **Is_input_available**

The transitions for IS_INPUT_AVAIABLE returning a 1 is given as:

$$\frac{A}{(Is\_input\_available(node\_number,\ vnum),\ s_1) \xrightarrow{node\_number,\ vnum} (1,\ s_2)} :: IS\_INPUT\_AVAILABLE$$

$$A \stackrel{def}{=\!=} \frac{(while\ (\underline{T})\ do\ (B),\ s_1) \xrightarrow{\underline{T}} (if\ (\underline{T})\ then\ (B;\ while\ (DS! = null)\ do\ (B)),\ s_2)}{(while(DS! = null)\ do\ B, s_1) \xrightarrow{\underline{T}} (1,\ s_2)}$$

$$B \stackrel{def}{=\!=} \frac{(DS.node\_number{=}node\_number\ ,\ s_1) \xrightarrow{\underline{T}} (\underline{T},\ s_1\{DS.node\_number{=}node\_number\})\quad (C, s_1) \xrightarrow{\underline{T}} (C, s_2)}{(if\ (DS.node\_number = node\_number)\ then\ C,\ s_1) \xrightarrow{node\_number} (\underline{T}, s_1)}$$

$$C \stackrel{def}{=\!=} \frac{(while\ (\underline{T})\ do\ (D),\ s_1) \xrightarrow{\underline{T}} (if\ (\underline{T})\ then\ (D;\ while\ (inlist! = null)\ do\ (D)),\ s_1)}{(while(inlist! = null)\ do\ D, s_1) \xrightarrow{\underline{T}} (1,\ s_1)}$$

$$D \stackrel{def}{=\!=} \frac{(inlist.vnum{=}vnum\ ,\ s_1) \xrightarrow{\underline{T}} (\underline{T},\ s_1\{inlist.vnum{=}vnum\})\quad (E, s_1) \xrightarrow{\underline{T}} (E, s_1)}{(if\ (inlist.vnum = vnum)\ then\ E,\ s_1) \xrightarrow{vnum} (\underline{T}, s_1)}$$

$$E \stackrel{def}{=\!=} \frac{(inlist.flag{=}1\ ,\ s_1) \xrightarrow{\underline{T}} (\underline{T},\ s_1\{inlist.flag{=}1\})\quad (availstatus{=}1, s_1) \xrightarrow{availstatus} (availstatus{=}1, s_2\{availstatus{=}1\})}{(if\ (inlist.flag = 1)\ then\ availstatus = 1,\ s_1) \xrightarrow{vnum} (\underline{T}, s_2\{availstatus = 1\})}$$

IS_INPUT_AVAIABLE returning a 0 is given as:

$$\frac{\dfrac{\dfrac{(\underline{F}, s_1) \xrightarrow{\underline{F}} (0, s_1)}{(while\ (\underline{F})\ do\ (DS{=}DS.next; A), s_1) \xrightarrow{\underline{F}} (if\ (\underline{F})\ then\ (A), s_1)}}{(while(DS!{=}null)\ do\ A, s_1) \xrightarrow{\underline{F}} (0, s_1)}}{(Is\_input\_available(node\_number,\ vnum),\ s_1) \xrightarrow{node\_number,\ vnum} (0,\ s_2)}$$

5. **Is_ok_to_send**

IS_OK_TO_SEND returning a 1:

$$\frac{A}{(Is\_ok\_to\_send(node\_number,\ vnum),\ s_1) \xrightarrow{node\_number,\ vnum} (1,\ s_2)} :: IS\_OK\_TO\_SEND$$

$$A \stackrel{def}{=\!=} \frac{(while\ (\underline{T})\ do\ (B),\ s_1) \xrightarrow{\underline{T}} (if\ (\underline{T})\ then\ (B;\ while\ (DS! = null)\ do\ (B)),\ s_2)}{(while(DS! = null)\ do\ B, s_1) \xrightarrow{\underline{T}} (1,\ s_2)}$$

$$B \stackrel{def}{=\!=} \frac{(DS.node\_number{=}Src\_node\_number\ ,\ s_1) \xrightarrow{\underline{T}} (\underline{T},\ s_1\{DS.node\_number{=}Src\_node\_number\})\quad (C, s_1) \xrightarrow{\underline{T}} (C, s_2)}{(if\ (DS.node\_number = Src\_node\_number)\ then\ C,\ s_1) \xrightarrow{node\_number} (\underline{T}, s_1)}$$

$$C \stackrel{def}{=\!=} \frac{(while\ (\underline{T})\ do\ (D),\ s_1) \xrightarrow{\underline{T}} (if\ (\underline{T})\ then\ (D;\ while\ (inlist! = null)\ do\ (D)),\ s_1)}{(while(outlist! = null)\, do\, D, s_1) \xrightarrow{\underline{T}} (1,\ s_1)}$$

$$D \stackrel{def}{=\!=} \frac{\overline{(outlist.vnum{=}vnum\ ,\ s_1) \xrightarrow{\underline{T}} (\underline{T},\ s_1\{outlist.vnum{=}vnum\})} \quad \overline{(E, s_1) \xrightarrow{\underline{T}} (E, s_1)}}{(if\ (outlist.vnum = vnum)\ then\ E,\ s_1) \xrightarrow{vnum} (\underline{T}, s_1)}$$

$$E \stackrel{def}{=\!=} \frac{\overline{(inlist.flag{=}1\ ,\ s_1) \xrightarrow{\underline{T}} (\underline{T},\ s_1\{outlist.counter{=}1\})} \quad \overline{(status{=}1, s_1) \xrightarrow{status} (status{=}1, s_2\{status{=}1\})}}{(if\ (outlist.counter = 1)\ then\ status = 1,\ s_1) \xrightarrow{vnum} (\underline{T}, s_2\{status = 1\})}$$

IS_OK_TO_SEND returning a 0 is given as:

$$\frac{\dfrac{\dfrac{\overline{(\underline{F}, s_1) \xrightarrow{\underline{F}} (0, s_1)}}{(while\ (\underline{F})\ do\ (DS{=}DS.next;A),\ s_1) \xrightarrow{\underline{F}} (if\ (\underline{F})\ then\ (A),\ s_1)}}{(while(DS\,!{=}\,null)\, do\, A, s_1) \xrightarrow{\underline{F}} (0, s_1)}}{(Is\_ok\_to\_send(node\_number,\ vnum),\ s_1) \xrightarrow{node\_number,\ vnum} (0,\ s_2)}$$

6. **Send**

Sending the data involves sending a data packet with the values of node number, variable number, type number and the actual value in string form. When the value is saved in the data structure, the counter is set to the number of nodes that are to receive the data by consulting the data matrix, DM, which consists of m number of rows and n number of columns where m is the number of variables where n is 3 (to store the variable number, its source node number and type number) + number of nodes.

Let i, j be two integers and let i=0 and j=3. Following is the set of transitions to find the the value of the counter using the data matrix, DM:

$$A \stackrel{def}{=\!=} \frac{(while\ (\underline{T})\ do\ (B),\ s_1) \xrightarrow{\underline{T}} (if\ (\underline{T})\ then\ (B;\ while\ (DS! = null)\ do\ (B)),\ s_2)}{(while(i <= m-1)\ do\ B, s_1) \xrightarrow{\underline{T}} (return\ counter,\ s_2\{counter\})}$$

$$B \stackrel{def}{=\!=} \frac{\overline{(if\ \underline{T}\ then\ (C, s_1) \xrightarrow{\underline{T}} (C, s_2))}}{(if\ (DM[i][0] = vnum)\ then\ C,\ s_1) \xrightarrow{vnum} (\underline{T}, s_1)}$$

$$C \stackrel{def}{=\!=} \frac{\overline{(if\ \underline{T}\ then\ (D, s_1) \xrightarrow{\underline{T}} (D, s_2))}}{(if\ ((DM[i][1] = node\_number) \wedge (DM[i][2] = type\_num))\ then\ D,\ s_1) \xrightarrow{vnum} (\underline{T}, s_1)}$$

$$D \stackrel{def}{=\!=} \frac{(while\ (\underline{T})\ do\ (E),\ s_1) \xrightarrow{\underline{T}} (if\ (\underline{T})\ then\ (E;\ while\ (j <= n-1)\ do\ (E)),\ s_2)}{(while(j <= n-1)\ do\ E, s_1) \xrightarrow{\underline{T}} (\underline{T},\ s_1)}$$

$$E \stackrel{def}{=\!=} \frac{(if\ (\underline{T})\ then\ (counter = counter + 1),\ s_1) \xrightarrow{\underline{T}} (\underline{T},\ s_1\{counter = counter + 1\})}{(if\ (DM[i][j] = 1)\ then\ (counter = counter + 1),\ s_1) \xrightarrow{DM[i][j],\ counter} (\underline{T},\ s_2\{counter = counter + 1\}}$$

Following is the set of transitions to place the data in the data structure and set the counter value successfully. That is, the function SEND is returning a 1:

$$\frac{}{(Send(src\_node\_num,\ vnum,\ type,\ data),\ s_1) \xrightarrow{src\_node\_num,\ vnum,\ type,\ data} (\underline{1},\ s_2)} :: SEND$$

$$A \stackrel{def}{=\!=} \frac{(while\ (\underline{T})\ do\ (B),\ s_1) \xrightarrow{\underline{T}} (if\ (\underline{T})\ then\ (B;\ while\ (DS! = null)\ do\ (B)),\ s_2)}{(while(DS! = null)\, do\, B, s_1) \xrightarrow{\underline{T}} (1,\ s_2)}$$

$$B \stackrel{def}{=\!=} \frac{\overline{(DS.node\_number{=}Src\_node\_number\ ,\ s_1) \xrightarrow{\underline{T}} (\underline{T},\ s_1\{DS.node\_number{=}Src\_node\_number\})} \quad \overline{(C,\ s_1) \xrightarrow{\underline{T}} (C,\ s_2)}}{(if\ (DS.node\_number = Src\_node\_number)\ then\ C,\ s_1) \xrightarrow{node\_number} (\underline{T},\ s_1)}$$

$$C \stackrel{def}{=\!=} \frac{(while\ (\underline{T})\ do\ (D),\ s_1) \xrightarrow{\underline{T}} (if\ (\underline{T})\ then\ (D;\ while\ (inlist! = null)\ do\ (D)),\ s_1)}{(while(inlist! = null)\, do\, D, s_1) \xrightarrow{\underline{T}} (1,\ s_1)}$$

$$D \stackrel{def}{=\!=} \frac{\overline{(inlist.vnum{=}vnum\ ,\ s_1) \xrightarrow{\underline{T}} (\underline{T},\ s_1\{inlist.vnum{=}vnum\})} \quad \overline{(E,\ s_1) \xrightarrow{\underline{T}} (E,\ s_1)}}{(if\ ((inlist.vnum = vnum) \wedge (inlist.flag = 0))\ then\ E,\ s_1) \xrightarrow{vnum} (\underline{T},\ s_1)}$$

$$E \stackrel{def}{=\!=} \frac{\dfrac{\overline{(G,\ s_1) \xrightarrow{G} (G,\ s_2\{G\})}}{(inlist.flag{=}0\ ,\ s_1) \xrightarrow{\underline{T}} (\underline{T},\ s_1\{G\})}}{(if\ (inlist.flag = 0)\ then\ G,\ s_1) \xrightarrow{vnum} (\underline{T},\ s_2\{G\})}$$

$$G \stackrel{def}{=\!=} \frac{}{(inlist.data = data, s_2) \xrightarrow{inlist.data,\ data} (-, s_2\{inlist.data = data\})} \quad H\ I$$

$$H \stackrel{def}{=\!=} \frac{}{(inlist.flag = 1, s_2) \xrightarrow{inlist.flag} (-, s_2\{inlist.flag = 1\})}$$

$$I \stackrel{def}{=\!=} \frac{}{(counter = counter - 1, s_2) \xrightarrow{counter} (-, s_2\{counter = counter - 1\})}$$

SEND returning a 0 is given as:

$$\frac{\dfrac{\dfrac{(\underline{F},\ s_1) \xrightarrow{\underline{F}} (0, s_1)}{(while\ (\underline{F})\ do\ (DS{=}DS.next;A),\ s_1) \xrightarrow{\underline{F}} (if\ (\underline{F})\ then\ (A),\ s_1)}}{(while(DS!{=}null)\, do\, A, s_1) \xrightarrow{\underline{F}} (0, s_1)}}{(Send(src\_node\_num,\ vnum,\ type,\ data),\ s_1) \xrightarrow{src\_node\_num,\ vnum,\ type,\ data} (0,\ s_2)}$$

4.4 Re-write Rules and LAM Codes for the Communication Part of LIPS

Section 4.2 describes the LAM codes for the computational part of LIPS using single-step re-write rules in an inductive fashion. This section extends the LAM codes to include the re-write rules for the communication part of LIPS and they are listed below:

1. **Push** a constant $\underline{n}$ of a specific data type σ in to the stack S is extended to include the following implicit data types.

   `channel, flag, node_number, node_name, vnum, vname, type_number, data, counter, data_packet, inlist, outlist, data_structure`

   It is assumed that the `data_packet, inlist, outlist, data_structure` constants occupy a single memory location in the LAM for easy understanding but

during real implementation they are handled independently. The re-write rule is written as follows:

$$\boxed{\underline{n} : C}\boxed{\text{S}}\boxed{\text{s}} \mapsto \boxed{\text{C}}\boxed{\underline{n} : \sigma : S}\boxed{\text{s}}$$

We mark the extended data types as implicit since

(a) They are used in asynchronous message passing and the programmer need not explicitly specify them while writing a LIPS program.

(b) They are created and used by the AMPS of LIPS at the time of executing the program.

2. **Fetch** a value from a memory location l and place it in the stack S. The data type of the value fetched is extended to include the additional data types which are `channel`, `flag`, `node_number`, `vnum`, `type_number`, `inlist`, `outlist`, `data_structure` and `data`. The re-write rule can be written as follows:

$$\boxed{l : C}\boxed{\text{S}}\boxed{\text{s}} \mapsto \boxed{\text{C}}\boxed{\underline{s(l)} : S}\boxed{\text{s}}$$

3. **Assignment** instruction has been extended in order to assign values of type `flag`, `node_number`, `vnum`, `type_number`, or `data` referred as P to a memory location l, i.e, $l := P$. The re-write rule is same as that for the usual assignment statement.

$$\boxed{l := P : C}\boxed{\underline{c} : S}\boxed{\text{s}} \mapsto \boxed{P : ASGNMNT(l) : C}\boxed{\text{S}}\boxed{\text{s}}$$

$$\boxed{ASGNMNT(l) : C}\boxed{\underline{n} : S}\boxed{\text{s}} \mapsto \boxed{P : ASGNMNT(l) : C}\boxed{\text{S}}\boxed{s\{l \mapsto \underline{c}\}}$$

4. **IF statement/alternate construct for a guarded process** can be specified using the following re-write rule:

$$\boxed{((fch_{i1} = \underline{T} \;\&\&\; fch_{i2} = \underline{T} \;\&\&\; \ldots \;\&\&\; fch_{im} = \underline{T})(P_{i1}; P_{i2}; \ \ldots\ ; P_{ik})) : C}\boxed{\underline{T} : S}\boxed{\text{s}}$$

$$\mapsto \boxed{(P_{i1}; P_{i2}; \ \ldots\ ; P_{ik}) : C}\boxed{\text{S}}\boxed{\text{s}}$$

fch_{i1}, fch_{i2}, ... fch_{im} are flags associated with the channels ch_{i1}, ch_{i2}, ... ch_{im} respectively and && specifies the logical AND.

5. **While statement for a node consisting of a set of guarded process** is an infinite loop. The re-write rule is given by using the existing if and while statements and it is given below:

$$\boxed{while\ \underline{T}\ do\ GIF : C}\boxed{\underline{T} : S}\boxed{\text{s}}$$

$$\mapsto \boxed{BR((GIF; while\ \underline{T}\ do\ GIF), EMPTY) : C}\boxed{\text{S}}\boxed{\text{s}}$$

The rewrite rule for GIF can be specified as below:

$$\|\,\mathrm{BR}(GIF_1, GIF_2, \ldots GIF_n) : C\,\|\,GIF_1 = \underline{T} : S\,\|\,\mathrm{s}\,\| \mapsto \|\,(P_{11}; P_{12}; \ \ldots\ ; P_{1k}) : C\,\|\,\mathrm{S}\,\|\,\mathrm{s}\,\|$$

$$\|\,\mathrm{BR}(GIF_1, GIF_2, \ldots GIF_n) : C\,\|\,GIF_2 = \underline{T} : S\,\|\,\mathrm{s}\,\| \mapsto \|\,(P_{21}; P_{22}; \ \ldots\ ; P_{2k}) : C\,\|\,\mathrm{S}\,\|\,\mathrm{s}\,\|$$

$$\vdots$$

$$\|\,\mathrm{BR}(GIF_1, GIF_2, \ldots GIF_n) : C\,\|\,GIF_n = \underline{T} : S\,\|\,\mathrm{s}\,\| \mapsto \|\,(P_{n1}; P_{n2}; \ \ldots\ ; P_{nk}) : C\,\|\,\mathrm{S}\,\|\,\mathrm{s}\,\|$$

where $GIF_i \stackrel{def}{=\!=} (fch_{i1} = \underline{T}\ \&\&\ fch_{i2} = \underline{T}\ \&\&\ \ldots\ \&\&\ fch_{im} = \underline{T})$ for $1 \leq i \leq n$.

6. The re-write rule for **Is_input_available** returning a 1 or 0 is given below

$$\|\,\mathit{Is\_input\_available}(\underline{\mathit{node\_number}}, \underline{\mathit{vnum}}) : C\,\|\,\mathrm{S}\,\|\,\mathrm{s}\,\|$$

$$\mapsto \|\,\mathrm{C}\,\|\,\underline{\mathit{node\_number}} : \underline{\mathit{vnum}} : \mathit{Is\_input\_available} : S\,\|\,\mathrm{s}\,\|$$

$$\mapsto \|\,\mathrm{C}\,\|\,\underline{i} : S\,\|\,\mathrm{s}\,\|$$

$$\mapsto \|\,\mathrm{C}\,\|\,S\,\|\,\mathrm{s}\{l \mapsto \underline{i}\}\,\|$$ where i can be either a 1 or a 0.

7. The re-write rule for **Is_ok_to_send** returning a 1 or 0 is given below:

$$\|\,\mathit{Is\_ok\_to\_send}(\underline{\mathit{node\_number}}, \underline{\mathit{vnum}}) : C\,\|\,\mathrm{S}\,\|\,\mathrm{s}\,\|$$

$$\mapsto \|\,\mathrm{C}\,\|\,\underline{\mathit{node\_number}} : \underline{\mathit{vnum}} : \mathit{Is\_ok\_to\_send} : S\,\|\,\mathrm{s}\,\|$$

$$\mapsto \|\,\mathrm{C}\,\|\,\underline{i} : S\,\|\,\mathrm{s}\,\|$$

$$\mapsto \|\,\mathrm{C}\,\|\,S\,\|\,\mathrm{s}\{l \mapsto \underline{i}\}\,\|$$ where i can be either a 1 or a 0.

8. The re-write rule for the **send** function is given below:

$$\|\,\mathit{send}(\underline{\mathit{data\_packet}}) : C\,\|\,\mathrm{S}\,\|\,\mathrm{s}\,\| \mapsto \|\,\mathrm{C}\,\|\,\underline{\mathit{data\_packet}} : \mathit{send} : S\,\|\,\mathrm{s}\,\|$$

$$\|\,C\,\|\,\mathrm{S}\,\|\,\mathrm{s}\,\| \mapsto \|\,\mathrm{C}\,\|\,\underline{\mathit{data\_packet}} : \mathit{send} : S\,\|\,\mathrm{s}\{l \mapsto \underline{i}\}\,\|$$ where i can be either a 1 or a 0.

4.4.1 Compilation of Communication Part of LIPS into the LAM codes

As discussed in Section 4.2.1, the function which takes a LIPS program expression and compiles it to a LAM code is given below:

$$[\![-]\!] : EXP \rightarrow LAMcodes$$

The LAM code for push, fetch and assignment operations are already stated in Section 4.2.1. The LAM codes for the rest of the communication part of LIPS is listed below:

1. **Guarded Process**

$$[\![\text{if } (fch_{i1} = \underline{T} \;\&\&\; fch_{i2} = \underline{T} \;\&\&\; ... \;\&\&\; fch_{im} = \underline{T}) \text{ then } (P_{i1}; P_{i2}; \ldots; P_{ik})]\!] \stackrel{def}{=\!=} [(fch_{i1} = \underline{T} \;\&\&\; fch_{i2} = \underline{T} \;\&\&\; ... \;\&\&\; fch_{im} = \underline{T})] : \text{BR}([\![(P_{i1}; P_{i2}; \ldots; P_{ik})]\!])$$

2. **Node**

$$[\![\text{while } \underline{T} \text{ do GIF }]\!] \stackrel{def}{=\!=} \text{WHILE}([\![\underline{T}]\!] \; do \; [\![GIF]\!])$$

where GIF is defined using the guarded process as below:
GIF $\stackrel{def}{=\!=}$ $GIF_1, GIF_2, \ldots, GIF_n$
For i where $1 \leq i \leq n$, GIF_i is defined as:

$$[\![\text{if } (fch_{i1} = \underline{T} \;\&\&\; fch_{i2} = \underline{T} \;\&\&\; ... \;\&\&\; fch_{im} = \underline{T}) \text{ then } (P_{i1}; P_{i2}; \ldots; P_{ik})]\!] \stackrel{def}{=\!=} [(fch_{i1} = \underline{T} \;\&\&\; fch_{i2} = \underline{T} \;\&\&\; ... \;\&\&\; fch_{im} = \underline{T})] : \text{BR}([\![(P_{i1}; P_{i2}; \ldots; P_{ik})]\!])$$

3. **Is_input_available**

$$[\![Is\_input\_available(node\_number, vnum)]\!] \stackrel{def}{=\!=} IS\_INPUT\_AVAILABLE(node\_number, vnum)$$

4. **Is_ok_to_send**

$$[\![Is\_ok\_to\_send(node\_number, vnum)]\!] \stackrel{def}{=\!=} IS\_OK\_TO\_SEND(node\_number, vnum)$$

5. **Is_ok_to_send**

$$[\![send(data\_packet)]\!] \stackrel{def}{=\!=} SEND(data\_packet)$$

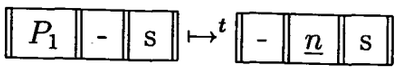
4.4.2 Correctness of the LAM

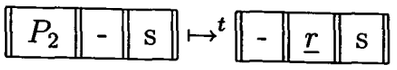
The dynamic model of LIPS programs can be compiled to low level LAM code. This section describes the correctness of the LAM with respect to the defined operational semantics. Theorem 1 is stated to show that the results of executing a LIPS expression specified using its operational semantics and the LAM code are identical.

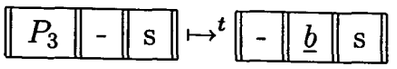
Theorem 1. For all

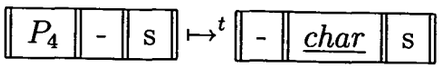
$$n \in \mathbf{Z}, r \in \mathbf{R}, b, \in \mathbf{B}, char \in \mathbf{CHAR}, str \in \mathbf{STR}, P_1 :: int,$$
$$P_2 :: real, P_3 :: bool, P_4 :: char, P_5 :: str, P_6 :: channel, P_7 :: flag, P_8 ::$$
$$node\_number, P_9 :: vnum, P_{10} :: type\_number, P_{11} :: data\_packet, P_{12} :: counter, P_{13} ::$$
$$vname, P_{14} :: inlist, P_{15} :: outlist, P_{16} :: data\_structure, and s, s_1, s_2 \in States$$

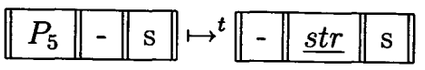
then we have,

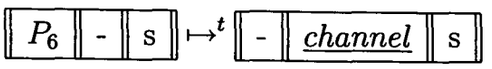
$(P_1, s) \Downarrow (\underline{n}, s)$ *iff* 

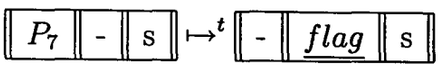
$(P_2, s) \Downarrow (\underline{r}, s)$ *iff* 

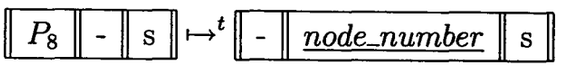
$(P_3, s) \Downarrow (\underline{b}, s)$ *iff* 

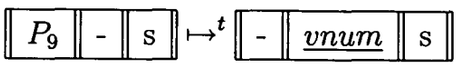
$(P_4, s) \Downarrow (\underline{char}, s)$ *iff* 

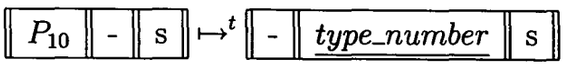
$(P_5, s) \Downarrow (\underline{str}, s)$ *iff* 

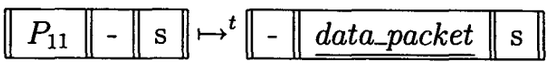
$(P_6, s) \Downarrow (\underline{channel}, s)$ *iff* 

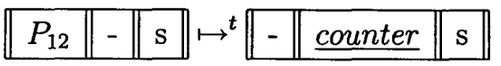
$(P_7, s) \Downarrow (\underline{flag}, s)$ *iff* 

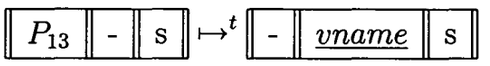
$(P_8, s) \Downarrow (\underline{node\_number}, s)$ *iff* 

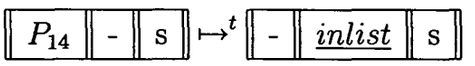
$(P_9, s) \Downarrow (\underline{vnum}, s)$ *iff* 

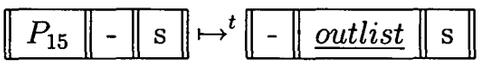
$(P_{10}, s) \Downarrow (\underline{type\_number}, s)$ *iff* 

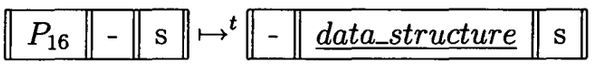
$(P_{11}, s) \Downarrow (\underline{data\_packet}, s)$ *iff* 

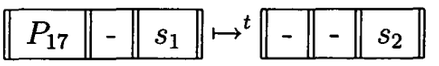
$(P_{12}, s) \Downarrow (\underline{counter}, s)$ *iff* 

$(P_{13}, s) \Downarrow (\underline{vname}, s)$ *iff* 

$(P_{14}, s) \Downarrow (\underline{inlist}, s)$ *iff* 

$(P_{15}, s) \Downarrow (\underline{data\_packet}, s)$ *iff* 

$(P_{16}, s) \Downarrow (\underline{data\_structure}, s)$ *iff* 

$(P_{17}, s_1) \Downarrow (empty, s_2)$ *iff* 

where $\mapsto^t$ refers the transitive closure[2] of $\mapsto$.

Theorem 1 is an extension of Crole's (2006) theorem to accommodate real, string, and character types of data and their respective operators.The proof of the theorem can be found in [Crole, 2006].

4.4.3 Executing the LAM Code

This section gives few examples to show the execution of the LAM code. The execution takes place in two steps:

[2]An extension or superset of a binary relation such that whenever (a, b) and (b, c) are in the extensions, (a, c) also in the extension. (Shukla,S,K,"transitive closure", in the Dictionary of Algorithms and Data Structures (online), Paul E. Black, ed., U.S. National Institute of Standards and Technology. (accessed 06/10/2006)Available from: http://www.nist.gov/dads/HTML/transitiveClosure.html)

Step 1: Compile the code.

Step 2: Execute the compiled code on the LAM.

Example 10. Let memory location x stores a floating point number 16.5. That is, s be the state of the LAM at which $s(x) = \underline{16.5}$.

Execute $20 + x$ on the LAM machine.

Compilation:

$$[\![\underline{20} + x]\!] = FETCH(x) : PUSH(\underline{20}) : OP(+)$$

Execution:

$$\boxed{\underline{20} + x \mid - \mid s} \mapsto \boxed{x : \underline{20} : + \mid - \mid s}$$

$$\mapsto \boxed{\underline{20} : + \mid \underline{16.5} \mid s} \quad \{\text{FETCH(x)}\}$$

$$\mapsto \boxed{+ \mid \underline{20} : \underline{16.5} \mid s} \quad \{\text{PUSH(20)}\}$$

$$\mapsto \boxed{- \mid \underline{36.5} \mid s} \quad \{20 + 16.5 = 36.5\}$$

Example 11. Let s be a state for which $s(x) = \underline{5}, s(y) = \underline{4}$ and let $\underline{10} * x + y$ has to be executed. Compilation:

$$[\![\underline{10} * x + y]\!] = FETCH(y) : FETCH(x) : PUSH(\underline{10}) : OP(*) : OP(+)$$

Execution:

$$\boxed{\underline{10} * x + y \mid - \mid s} \mapsto \boxed{y : x : \underline{10} : * : + \mid - \mid s}$$

$$\mapsto \boxed{y : \underline{10} : * : + \mid \underline{5} \mid s} \quad \{\text{FETCH(x)}\}$$

$$\mapsto \boxed{y : * : + \mid \underline{10} : \underline{5} \mid s} \quad \{\text{PUSH(10)}\}$$

$$\mapsto \boxed{y : + \mid \underline{50} \mid s} \quad \{10 * 5 = 50\}$$

$$\mapsto \boxed{+ \mid \underline{4} : \underline{50} \mid s} \quad \{\text{FETCH(y)}\}$$

$$\mapsto \boxed{- \mid \underline{54} \mid s} \quad \{50 + 4 = 54\}$$

Example 12. Let s be a state for which $s(x) = 1$. Execute $x = x - \underline{1}$.

Compilation:

$[\![x = x - \underline{1}]\!]$

Execution:

Step1: Evaluate the program expression. $FETCH(x) : PUSH(\underline{1}) : OP(-)$

Step2: Assign the result to x.

$$\boxed{x = x - \underline{1} \mid - \mid s} \mapsto \boxed{x - \underline{1} : ASGNMNT(x) \mid - \mid s}$$

$$\mapsto \boxed{\underline{1} : x : - : ASGNMNT(x) \mid - \mid s}$$

$\mapsto$ | $x : - : ASGNMNT(x)$ | $\underline{1}$ | s | {PUSH(1)}

$\mapsto$ | $- : ASGNMNT(x)$ | $\underline{1} : \underline{1}$ | s | {FETCH(x)}

$\mapsto$ | $ASGNMNT(x)$ | $\underline{1} : \underline{1} : -$ | s | {Apply operator}

$\mapsto$ | $ASGNMNT(x)$ | $\underline{0} -$ | s | {Assign result to x}

$\mapsto$ | - | - | $s\{x \mapsto 0\}$ |

Example 13. Let s be a state for which $s(x) = 1$. Execute the following if statement:
if $l \geq 0$ *then* $l := l - \underline{1}$ *else empty*

1. Compilation

$l \geq 0 : BR(l := l - \underline{1}, empty)$

2. Execution

| $l \geq 0 : BR(l := l - \underline{1}, empty)$ | - | s | $\mapsto$ | $0 : x \geq BR(l := l - \underline{1}, empty)$ | - | s |

$\mapsto$ | $l :\geq BR(l := l - \underline{1}, empty)$ | $\underline{0}$ | s | {PUSH(0)}

$\mapsto$ | $\geq BR(l := l - \underline{1}, empty)$ | $\underline{1} : \underline{0}$ | s | {FETCH(l)}

$\mapsto$ | $BR(l := l - \underline{1}, empty)$ | $\underline{T}$ | s | {Branch when $l \geq 0$}

$\mapsto$ | $l := l - \underline{1}$ | - | s |

$\mapsto$ | $l - \underline{1} : ASGNMNT(l)$ | - | s |

$\mapsto$ | $l : x : - : ASGNMNT(l)$ | - | s |

$\mapsto$ | $l : x : - : ASGNMNT(l)$ | - | s |

$\mapsto$ | $l : - : ASGNMNT(l)$ | - | s | {PUSH(x)}

$\mapsto$ | $- : ASGNMNT(l)$ | $\underline{1} : \underline{1}$ | s | {FETCH(l)}

$\mapsto$ | $ASGNMINT(l)$ | $\underline{0}$ | s | {Assign result to y}

$\mapsto$ | - | - | $s\{l \mapsto 0\}$ |

Example 14. Consider the LIPS program to solve the Vending Machine Problem given in Appendix A.1. Following is the code segment of a guard waiting for the values of coin and button:

```
[coin, button]=>{
                print("machine has received ", coin, "p");
                //set drksig
                drkSig=true;
}
```

Let `fcoin` and `fbutton` are the two flags associated with `coin` and `button` channels and and && specifies the logical AND. Execute the guard [coin, button] on the LAM machine. Compilation:

$$[\![\text{if } (fcoin = \underline{T} \;\&\&\; fbutton = \underline{T}) \text{ then}$$
$$(print("machine\ has\ received", coin, "p"); drkSig = true)\ ]\!]$$
$$\stackrel{def}{=\!=}(fcoin = \underline{T} \;\&\&\; fbutton = \underline{T}) :$$
$$\mathrm{BR}([\![\ (print("machine\ has\ received", coin, "p"); drkSig = true)\ ]\!])$$

The above guarded process can be specified using the following re-write rule:

Execution:

$$\| ((fcoin = \underline{T} \;\&\&\; fbutton = \underline{T})(print("machine\ has\ received", coin, "p"); drkSig = true)) : C \| S \| \mathrm{s} \|$$
$$\mapsto \| ((\&\&\; fbutton = \underline{T})(print("machine\ has\ received", coin, "p"); drkSig = true)) : C \| fcoin = \underline{T} : S \| \mathrm{s} \|$$
$$\mapsto \| ((\&\&)(print("machine\ has\ received", coin, "p"); drkSig = true)) : C \| fbutton = \underline{T} : fcoin = \underline{T} : S \| \mathrm{s} \|$$
$$\mapsto \| ((print("machine\ has\ received", coin, "p"); drkSig = true)) : C \| \&\& : fbutton = \underline{T} : fcoin = \underline{T} : S \| \mathrm{s} \|$$
$$\mapsto \| ((print("machine\ has\ received", coin, "p"); drkSig = true)) : C \| \underline{T} : S \| \mathrm{s} \|$$
$$\mapsto \| ((drkSig = true)) : C \| print("machine\ has\ received", coin, "p") : S \| \mathrm{s} \|$$
$$\mapsto \| - : C \| (drkSig = true) : print("machine\ has\ received", coin, "p") : S \| \mathrm{s} \|$$
$$\mapsto \| - : C \| - : S \| \mathrm{s}\{(\mathrm{drkSig{=}true}){:}\mathrm{print}(\text{"machine has received "}, \mathrm{coin}, \text{"p"})\} \|$$

4.5 Summary

Operational semantics provide a clear mathematical formulation of the meaning of individual language constructs and of the language itself. The underlying research work involves the implementation of LIPS language and the definition of operational semantics which can be used to refine the LIPS compiler. The defined semantics can also be used for the specification and verification of LIPS programs. This chapter gives a mathematical model for the executable statements of a LIPS program and justifies the development of formal semantics for LIPS. A mixed approach has been adopted in which

1. The operational semantics for the computational part of LIPS has been defined using big-step semantics

2. The developed big-step semantics has been extended to include the communication part of LIPS.

An abstract machine called LIPS Abstract Machine (LAM) which works on the basis of single-step rewrite rules has been defined. The code needed for the operational semantics has been verified for its correctness against the LAM that describes the executional behaviour in this context. The communication schema derived not only describe the asynchronous communication that takes place in LIPS but also can serve as a reference for implementers of the language.

Chapter 5

Operational Semantics for SACS

Process algebra is considered as a formal framework to model concurrent systems of interacting processes and their behaviour. Few of the well known process algebraic tools include Communicating Sequential Processes (CSP) [Hoare, 1978], Calculus of Communicating Systems (CCS) [Milner, 1982], Synchronous Calculus of Communicating Systems (SCCS) [Gray, 2000], and Language of Temporal Ordering Specifications (LOTOS) [Logrippo et al., 1990]. Since the development of CCS many extensions have been proposed to model different aspects of concurrent processing [Galpin, 1998]. Specification of Asynchronous Communicating Systems (SACS) [Bavan and Illingworth, 2000, Bavan et al., 2007a] is one of them. SACS is a point-to-point message passing system which is an asynchronous variant of SCCS developed to specify the communicating part of LIPS. The main objective of SACS is to separate the specification of communication from the computation part of LIPS programs so that they can proceed independently.

The behaviour of process algebra can be described using Structural Operational Semantics (SOS) which is defined using Labelled Transition Systems (LTS). This can be used to study semantic equivalences. Two programs are said to be semantically equivalent if they cannot be distinguished. Semantic equivalences are used to abstract the internal structure of the programs that cannot be otherwise observed. They also provide a successful method to verify program behaviour [Gray, 2000]. Verifying a program means to show that the program is behaviourally equal to its specification.

In this chapter we describe the Structural Operational Semantics and analyse bisimulation equivalence properties for SACS. Since LIPS is designed based on SACS and the communication part is implemented using Asynchronous Message Passing System (AMPS) described in Chapter 3, it is necessary to verify that both are equivalent. The proof of equivalence of SACS and AMPS is demonstrated using an example.
This Chapter is organised as follows:

- Section 5.1: An introduction to SACS specification.
- Section 5.2: Structural Operational Semantics for SACS using its syntactic cate-

gories.

- Section 5.3: Bisimulation and observational equivalence for SACS.
- Section 5.4: Proof of equivalence for the SACS and AMPS.
- Section 5.5: Summary.

5.1 SACS - An Introduction

SACS works according to the four design rules stated in the Literature Review Chapter (See Section 2.3.1 in Chapter 2). These rules have their origin in LIPS and assist in specifying the communication part of LIPS. The specifications generated using SACS are defined using a standard template which is shown below:

```
Process_agent_1 = input_ch1_1[[.|:|+]input_ch1_n]:Process_agent_1Bdy_1
                  +...+ [input_chk_1[[.|:|+]input_chk_n]:Process_agent_1Bdy_k]
Process_agent_1Bdy_i = output_ch1_1[[.|:|+]output_ch1_n]:[@|Process_agent_i]
where i ranges from 1 to k.
```

Definitions for the operators used in SACS - TEMPLATE are shown in Table 5.1

Table 5.1: Operators used in SACS

'.'	simultaneous AND operator
':'	sequential AND operator
'+'	OR operator
{ }	repetition
( )	selection
[ ]	optional

To describe the SACS specifications the following two examples are being used in this section:

- Finding the area under a curve using Simpson's rule
- Vending machine problem

Both of the examples are described in Chapter 3.

Example 1. Finding the Area Under a curve Using Simpson's Rule:

Consider the example of calculating the area under a curve $y = f(x)$ using Simpson's rule described in Section 3.2. One possible SACS specification for this scenario is shown in Figure 5.1.

```
// Host sends width and segment to Area node
Host = width!:segment[0..2]!:Host+1:result?:0

//Area receives width and segment number. It calculates the value of
// the area corresponding to a particular with and segment number and
// sends it to the Summer node
Area = 1:width?:segment[0..2]?:area[0..2]!:Area

//Summer receives area values and add them up. The added value is
// sent as result to the Host node
Summer = 1:area[0..2]?:result!:Summer

Where,
            System = Host x Area x Summer
```

Figure 5.1: SACS specification for Simpson's Rule.

Example 2. Vending Machine Problem

Consider the vending machine example described in Section 3.5 which requires a customer to insert a coin and press a button after which the machine will deliver a drink. The behaviour of the vending machine can be illustrated using four processes namely `INIT`, `CUSTOMER`, `MACHINE_INTERFACE` and `MACHINE_INTERNALS` which are briefly described below:

- `INIT`: This process outputs a `trayEmpty` signal and terminates. This signal infers that the tray is empty. Without such a signal, the vending machine would deliver the drink without checking whether the previously delivered drink has been removed. This may result in a vending machine that delivers drinks one above the other.

- `CUSTOMER`: This process requires the customer to insert a `coin` and press a `Button` and waits for the drink to be `delivered`. It terminates after receiving the drink.

- `MACHINE_INTERFACE`: This process has been included in the vending machine example to separate the user interface from the internal working of the vending machine. This process receives the `coin` and `Button` as inputs. On receiving these inputs, it outputs a signal, `drkSig`, to let the machine internals know that the drink can be prepared.

- `MACHINE_INTERNALS`: This is a process which actually prepares and delivers the drink after receiving `drkSig` and `trayEmpty` signals from `MACHINE_INTERFACE` and `INIT` respectively.

The SACS specification for the vending machine example is shown in Figure 5.2.

```
// Define the system
  SYSTEM = INIT x CUSTOMER x MACHINE_INTERFACE x
                    MACHINE_INTERNALS

//Init simply sends a trayEmpty once and terminate
INIT = trayEmpty!:@

//customer can insert a coin and press the button
CUSTOMER = coin!:Button!:CUSTOMER + 1:deliver?:@

//machine interface receives coin and button and delivers drkSig
MACHINE_INTERFACE =
         1:coin?:button?:drkSig!:MACHINE_INTERFACE

// machine internal makes the drink only when it receives trayEmpty
//and drkSig
MACHINE_INTERNALS =
 (drkSig?.trayEmpty?):deliver!:MACHINE_INTERNALS
```

Figure 5.2: SACS specification for the vending Machine Problem.

5.2 Structural Operational Semantics for SACS

The first step in describing the SOS is to define a formal syntax which is expressed using the syntactic categories. The syntactic categories for SACS are described in this section.

5.2.1 Syntactic Categories of SACS

The basic elements of SACS are

1. **(channel, port) names** - `a, b, c, ...`
   The ports are the observable parts of an agent/process which support either sending or receiving of information. Channels are individual paths through which data (signals) can flow.

2. **co-names** - `a?, b?, c?, ...` and `a!, b!, c!, ...`
   Co-names are derived from the names and are used to specify the input and output channels.

`a?, b?, c?, ...`	are the input channels
`'?'`	denotes that the channel is waiting for input.
`a!, b!, c!, ...`	are the output channels
`'!'`	denotes sending an output.

   Ports having the same names synchronise/interact.

3. **Idle action** τ ::= `1`, δ

   Idle action is denoted by either a `1` or δ.

   `1` denotes an idle event which is always suffixed with ':' and written as `'1:'`. Idle event is introduced to enable the send/receive pairs to synchronise.

   δ introduces a delay. A delay is introduced before a value is received or sent by the channel. Let `a` be either an input channel or output channel. The delay introduced during the process of receiving or sending a value through the channel is denoted as δa. δa is replaced by `1:`δ`a+a` (e.g. δ`a!` = `1:`δ`a+a!`).

4. **prefix** ::= τ`|a?[.|:|+]b?[.|:|+]c?[.|:|+]...`

   A prefix is a guard where `a, b, c,...` are the input ports. The guard becomes true only when all the input ports of a specific guard have new values.

5. **Process_agent/node** ::= `@| GP1 + GP2 + GP3 + ... + GPk`

   A Process_agent/node consists of a set of guarded processes `GP1, GP2, GP3,...,GPk`. The `@` denotes inaction - equivalent to stop in CCS and CSP.

6. **Guarded process**: `GP1 ::=` α `P1`

   A guarded process has a guard and a statement block.The guard α is the prefix which is denoted by τ`|a?[.|:|+]b?[.|:|+]c?[.|:|+]...`. When α becomes true it will perform the process `P1`. The process consists of a process body `P` followed by a set of output channels (if there are any generated) which is denoted as `output_ch1_1[[.|:|+]output_ch1_n]:(@|Process_agent)`.

7. **System** ::= `P x Q x R x ...`

   A LIPS program consists of a set of Process_agents/nodes `P, Q, R, ....` `x` is the concurrency operator. It is the same as that of `P|Q` in CCS and `P||Q` in CSP except that `P x Q` indicates asynchronous communication.

8. Operators:
   - `'x'` concurrency operator
   - `':'` sequential AND operator
   - `'.'` simultaneous AND operator
   - `'+'` OR operator

The syntactic categories of the SACS are listed in Table 5.2.

The set of input channels, α, and the set of output channels, β, of SACS are

i. Disjoint ($\alpha \cap \beta = \phi$)

ii. Bijection via the complement function

   $\alpha, \beta \subseteq \kappa$ such that

   $a!? = a$ and $a? \neq a!$ for all a $\in \kappa, a? \in \alpha$, and $a! \in \beta$.

Table 5.2: Syntactic Categories of the SACS

Set of channels	$\kappa \stackrel{def}{=\!=}$ {a, b,...} where κ is finite
Set of input channels	$\alpha \stackrel{def}{=\!=}$ {a?, b?,...} where a, b,... $\in \kappa$ and α is finite.
Set of output channels	$\beta \stackrel{def}{=\!=}$ {a!, b!,...} where a, b,... $\in \kappa$ and β is finite.
Inaction	$\iota \stackrel{def}{=\!=}$ @
Silent action	$\tau \stackrel{def}{=\!=} \{1, \delta\}$ where 1 is idle event δ is the delay.
Action (input and output)	$\Omega \stackrel{def}{=\!=} \{\alpha \cup \beta \cup \tau \cup \iota\}$
Operator Concurrency Operator Choice Operator	$\bigcirc \stackrel{def}{=\!=} \{.,:,+\}$ where '.' is simultaneous AND, ':' is sequential AND X $\stackrel{def}{=\!=}$ {x} + $\stackrel{def}{=\!=}$ {+}
Guard/input ports	$\eta \stackrel{def}{=\!=} \iota \bigcirc \tau \bigcirc \alpha_1 \bigcirc \alpha_2 \bigcirc ... \bigcirc \alpha_n$ where $\alpha_1, \alpha_2, ..., \alpha_n \in \alpha$ and n is a positive integer
List of output ports	$\sigma \stackrel{def}{=\!=} \iota \bigcirc \tau \bigcirc \beta_1 \bigcirc \beta_2 \bigcirc ... \bigcirc \beta_m$ where $\beta_1, \beta_2, ..., \beta_m \in \beta$ and m is a positive integer
Guarded Process	GP $\stackrel{def}{=\!=} \eta$Proc where Proc is a process body which may generate a list of output ports
Node	N $\stackrel{def}{=\!=} GP1 + GP2 + ... + GP_k$ where k is a positive integer
Set of Processes/Nodes	$\Re \stackrel{def}{=\!=} \{P, Q, ...\}$ where P, Q,...$\in$ N
System	$\triangle \stackrel{def}{=\!=} \Re_1 X \Re_2 X ... X \Re_p$ $\Re_1, \Re_2, ... \Re_p \in \Re$ and p is a positive integer

The set of $\alpha \cup \beta$ is a visible set of actions. Let \$ be the operator used to denote recursion, \$P denotes recursion where P is the recursive process. That is,

$$\$P = P \text{ x } \$P = P \text{ x } P \text{ x } \$P = P \text{ x } P \text{ x } ... \text{ x} \$P$$

where P can be repeated as many times as needed and the number of repetitions is finite. Based on the syntactic categories the SOS for SACS can be defined. This is done using transition relations called Labelled Transition Systems (LTS). The LTS for SACS is described in the following section.

5.2.2 Labelled Transition System Configurations for SACS

We gave the definition for the Labelled Transition System in Section 4.3.4 of Chapter 4. A transition can be specified as $s \xrightarrow{k} s'$.

We define the transitions of SACS using the following inference rules:

1. **Guard/input ports**:
   For a guard η consisting of k input channels $\alpha_1, \alpha_2, ..., \alpha_n \in \alpha$ is specified as:

$$\frac{}{\alpha_1 \bigcirc \alpha_2 \bigcirc ... \bigcirc \alpha_n \longrightarrow \alpha_1 \bigcirc \alpha_2 \bigcirc ... \bigcirc \alpha_n} :: \eta$$

   The operator $\bigcirc$ can be either '.' or ':' or '+'. The SOS for any two input channels $\alpha_1, \alpha_2 \in \alpha$ is given below:

   (a) Simultaneous AND:

$$\frac{}{\alpha_1 . \alpha_2 \longrightarrow \alpha_i . \alpha_j} :: SimultaneousAND$$

   where i and j can be either 1 or 2.

   (b) Sequential AND:

$$\frac{}{\alpha_1 : \alpha_2 \longrightarrow \alpha_1 : \alpha_2} :: SequentialAND$$

   (c) OR:

$$\frac{}{\alpha_1 + \alpha_2 \longrightarrow \alpha_i} :: OR$$

   where i can be either 1 or 2 at a specific time.

2. **Guarded Process (GP)**:

$$\frac{}{\eta \texttt{Proc} \xrightarrow{\eta} \texttt{Proc}'\sigma} :: GP$$

   where η is a guard and σ is the set of output channels generated by the process body `Proc`.

3. **Node** $\Re = GP_1 + GP_2 + ... + GP_k$

$$\frac{\forall i : 1 \leq i \leq k : \eta_i \ \texttt{Proc}_i \xrightarrow{\eta_i} \texttt{Proc}'_i \sigma_i}{\sum_{i=1}^{k} \eta_i \texttt{Proc}_i \xrightarrow{for\ any\ i=1\ to\ k, \eta_i} \texttt{Proc}'_i \sigma_i} :: \Re$$

4. **Concurrency Composition** ($\Re_1$ x $\Re_2$)

$$\frac{\Re_1 \xrightarrow{\mu} \Re_1' \quad \Re_2 \xrightarrow{\mu} \Re_2'}{\Re_1 \text{ x } \Re_2 \xrightarrow{\mu} \Re_1' \text{ x } \Re_2'} :: \Re_1 \text{ X } \Re_2$$

where μ is a guard comprising of set of input channels.

5. **System** ($\Re_1$ x $\Re_2$ x ... x $\Re_m$) A system ∇ is defined as $\Re_1$ x $\Re_2$ x ... x $\Re_m$ where m is a positive integer.

$$\frac{\Re_1 \xrightarrow{\mu} \Re_1' \quad \Re_2 \xrightarrow{\mu} \Re_2' \ldots \Re_m \xrightarrow{\mu} \Re_m'}{(\Re_1 \text{ x } \Re_2 \text{ x } \ldots \text{ x} \Re_m, s_1) \xrightarrow{\mu} (\Re_1' \text{ x } \Re_2' \text{ x } \ldots \text{ x } \Re_m', s_2)} :: \nabla$$

In the rest of the section we demonstrate through the examples how the inference rules can be used to specify SOS.

Example 3. Vending Machine:

Consider the vending machine problem described in Section 5.1 for which the SACS specification is shown in Figure 5.2. The LTS used to express the SOS are given below:

$$\frac{N_1 \quad N_2 \quad N_3 \quad N_4}{INIT \text{ x } CUSTOMER \text{ x } MACHINE\_INTERFACE \text{ x } MACHINE\_INTERNALS \xrightarrow{AMPS} INIT' \text{ x } CUSTOMER' \text{ x } MACHINE\_INTERFACE' \text{ x } MACHINE\_INTERNALS'}$$

$$N_1 = \frac{}{\dfrac{INIT \xrightarrow{0} INIT' \cdot trayEmpty! : @}{INIT \xrightarrow{0} INIT'}} :: INIT$$

$$N_2 = \frac{N_{2_1}}{CUSTOMER \xrightarrow{deliver?} CUSTOMER'} :: CUSTOMER$$

$$N_{2_1} = \frac{X1 \quad X2}{CUST\_AGT1 + 1 : deliver? \cdot CUST\_AGT2 \xrightarrow{1} CUST\_AGT1' : coin! : Button! + CUST\_AGT2'} :: CUST\_AGT1$$

$$X1 = \frac{}{CUST\_AGT1 \xrightarrow{0} CUST\_AGT1' : coin! : Button!} :: CUST\_AGT1$$

$$X2 = \frac{}{1 : deliver? \cdot CUST\_AGT2 \xrightarrow{deliver?} CUST\_AGT2' : @} :: CUST\_AGT2$$

$$N_3 = \frac{N_{3_1}}{MACHINE\_INTERFACE \xrightarrow{1:coin?:Button?} MACHINE\_INTERFACE'} :: MACHINE\_INTERFACE$$

$$N_{3_1} = \frac{}{1 : coin? : Button? : MAC\_INT\_AGT} :: MAC\_INT\_AGT$$
$$\xrightarrow{1 : coin? : button?} MAC\_INT\_AGT' : drkSig!$$

$$N_4 = \frac{N_{4_1}}{MACHINE\_INTERNALS} :: MACHINE\_INTERNALS$$
$$\xrightarrow{drkSig? : trayEmpty?} MACHINE\_INTERNALS'$$

$$N_{4_1} = \frac{}{drkSig? : trayEmpty? : MAC\_INTL\_AGT \xrightarrow{drkSig? : trayEmpty?}} :: MAC\_INTL\_AGT$$
$$MAC\_INTL\_AGT' : deliver!$$

Consider two different representations of the vending machine described in Section 5.1. We change the problem so that the customer can order a coffee or tea by performing either one of the following actions:

- insert a coin and press the coffee button to order coffee
- insert a coin and press the tea button to order tea.

We consider two different implementations for this modified scenario and name them VENDING_MACHINE1 and VENDING_MACHINE2. For each of these implementations SACS and SOS specifications are described below:

Example 4. VENDING_MACHINE1:

This representation uses two machine interfaces, `MAC_INT1` and `MAC_INT2`, and two machine internals, `COF_MAC` and `TEA_MAC`, one to make coffee and the other one to make tea separately. Other processes involved are `INIT` and `CUST`. These processes are briefly described below:

i. `INIT`: This process outputs a signal called trayEmpty and terminates. It functions similar to the vending machine shown in Example 5.1.

ii. `CUST`: This process is implemented as below:

- if it receives a `coin`, and the coffee button, `C_Button`, as input it sends them as output to MAC_INT1 and waits for `Coffee` to be delivered
- if it receives a `coin`, and the tea button, `T_Button`, as input it sends them as output to MAC_INT2 and waits for `Tea` to be delivered.

This process terminates after receiving the drink.

iii. MAC_INT1: This process acts as a user interface. It receives a coin and the C_Button and outputs Deliver_Coffee signal.

iv. MAC_INT2: This process acts as a user interfaces similar to MAC_INT1. It receives a coin and the T_Button as input and outputs Deliver_Tea signal.

v. COF_MAC: This is one of the machine internals which makes coffee. It delivers Coffee after receiving trayEmpty signal from INIT and Deliver_Coffee signal from MAC_INT1.

vi. TEA_MAC: This is also one of the machine internals which makes tea. It delivers Tea after receiving trayEmpty from INIT and Deliver_Tea from MAC_INT2.

A diagrammatic representation of the VENDING_MACHINE1 is shown Figure 5.3. Dotted lines represent the optional actions which can take place at a particular time.

Example: For a particular instance, CUST will output either coin and C_Button or coin and T_Button depending on the input signals received from the customer.

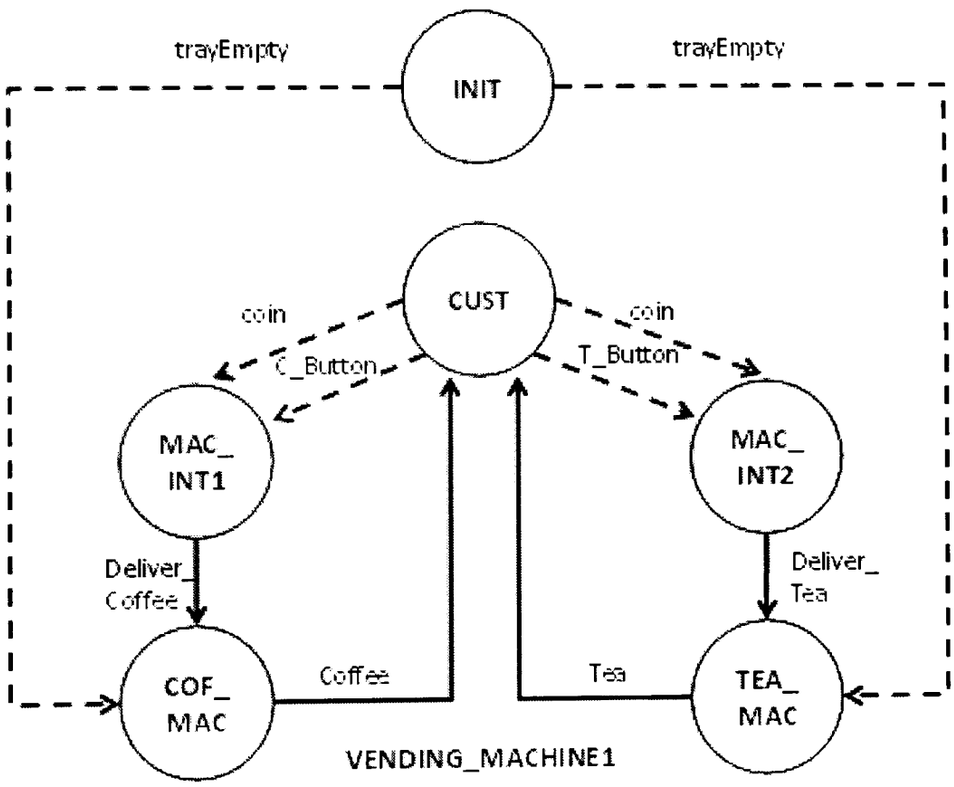


Figure 5.3: VENDING_MACHINE1.

Figure 5.4 shows the SACS specification for VENDING_MACHINE1.

```
//Init simply sends a trayEmpty once and terminate
INIT = trayEmpty!:@

//customer inserts coin & presses coffee button or inserts coin & presses tea button to
//receive a coffee or tea respectively
CUST = ((coin!:C_Button!) + (coin!:T_Button!)):CUST
                          + 1:(Coffee? + Tea?):@

//machine interface1 sends deliver coffee signal after receiving coin & C_Button
MAC_INT1 = 1:coin?:C_Button?:Deliver_Coffee!:MAC_INT1

//machine interface2 sends deliver tea signal after receiving coin & T_Button
MAC_INT2 = 1:coin?:T_Button?:Deliver_Tea!:MAC_INT2

//coffee machine makes coffee after deliver coffee signal and trayEmpty signal
COF_MAC = (Deliver_Coffee?.trayEmpty?):Coffee!:COF_MAC

//Tea machine makes tea after deliver tea signal and trayEmpty signal
TEA_MAC = (Deliver_Tea?.trayEmpty?):Tea!:TEA_MAC

//Define the system
SYSTEM =
INIT x CUST x MAC_INT1 x MAC_INT2 x COF_MAC x TEA_MAC
```

Figure 5.4: SACS specification for VENDING_MACHINE1.

The LTS used to express the SOS derived from the inference rules for VENDING_MACHINE1 is given below:

$$\frac{N_1 \quad N_2 \quad N_3 \quad N_4 \quad N_5 \quad N_6}{INIT \,\mathrm{x}\, CUST \,\mathrm{x}\, MAC\_INT1 \,\mathrm{x}\, MAC\_INT2 \,\mathrm{x}\, COF\_MAC \,\mathrm{x}\, TEA\_MAC}$$

$$N_1 = \frac{}{\dfrac{INIT \xrightarrow{0} INIT' \cdot trayEmpty! : @}{INIT \xrightarrow{0} INIT'}} :: INIT$$

$$N_2 = \frac{N_{2_1} \quad N_{2_2}}{\dfrac{CUST\_AGT1 + CUST\_AGT2 \xrightarrow{coffee? + tea?} CUST\_AGT1' + CUST\_AGT2'}{CUST \xrightarrow{coffee? + tea?} CUST'}} :: CUST$$

$$N_{2_1} = \frac{N_{2_{11}}}{CUST\_AGT1 \xrightarrow{0} CUST\_AGT1' : (coin! : C\_Button! + coin! : T\_Button!)} :: CUST\_AGT1$$

or

$$N_{2_1} = \frac{N_{2_{12}}}{CUST\_AGT1 \xrightarrow{0} CUST\_AGT1' : (coin! : C\_Button! + coin! : T\_Button!)} :: CUST\_AGT1$$

$$N_{2_{11}} = \frac{}{CUST\_AGT1 \xrightarrow{0} CUST\_AGT1' : coin! : C\_Button!} :: CUST\_AGT1$$

$$N_{2_{12}} = \frac{}{CUST\_AGT1 \xrightarrow{0} CUST\_AGT1' : coin! : T\_Button!} :: CUST\_AGT1$$

$$N_{2_2} = \frac{\dfrac{}{coffee? : CUST\_AGT2 \xrightarrow{coffee?} CUST\_AGT2' : @}}{(coffee? + tea?) : CUST\_AGT2 \xrightarrow{(coffee? + tea?)} CUST\_AGT2' : @} :: CUST\_AGT2$$

or

$$N_{2_2} = \frac{\dfrac{}{tea? : CUST\_AGT2 \xrightarrow{tea?} CUST\_AGT2' : @}}{(coffee? + tea?) : CUST\_AGT2 \xrightarrow{(coffee? + tea?)} CUST\_AGT2' : @} :: CUST\_AGT2$$

$$N_3 = \frac{}{1 : coin? : C\_Button? : MAC\_INT1 \xrightarrow{1 : coin? : C\_Button?} MAC\_INT1' : Deliver\_Coffee!} :: MAC\_INT1$$

$$N_4 = \frac{}{1 : coin? : T\_Button? : MAC\_INT2 \xrightarrow{1 : coin? : T\_Button?} MAC\_INT2' : Deliver\_Tea!} :: MAC\_INT2$$

$$N_5 = \frac{}{Deliver\_Coffee? : trayEmpty? : COF\_MAC \xrightarrow{Deliver\_Coffee? : trayEmpty?} COF\_MAC' : Coffee!} :: COF\_MAC$$

$$N_6 = \frac{}{Deliver\_Tea? : trayEmpty? : TEA\_MAC \xrightarrow{Deliver\_Tea? : trayEmpty?} TEA\_MAC' : Tea!} :: TEA\_MAC$$

Example 5. VENDING_MACHINE2:

The second implementation of the vending machine, `VENDING_MACHINE2`, uses one machine interface and two machine internals, to make coffee and tea separately. The processes involved in this implementation are `INIT`, `CUSTOMER`, `MAC_INT`, `COF_MAC` and `TEA_MAC`. They are briefly described below:

i. `INIT`: It is the same as that of the `VENDING_MACHINE1` shown in Example 4.

ii. `CUST`: When this process receives the combination `coin` and `C_Button` or `T_Button`, it sends them as output to `MAC_INT` and waits for the drink (either `Coffee` or `Tea`) to be delivered. This process terminates after receiving the drink.

iii. MAC_INT: This process is a user interface. When it receives the coin and C_Button, it outputs Deliver_Coffee signal. When it receives coin and T_Button, it outputs Deliver_Tea signal.

iv. COF_MAC: This process is one of the machine internals to make coffee. It delivers Coffee after receiving trayEmpty signal from INIT and Deliver_Coffee signal from MAC_INT.

v. TEA_MAC: This process is the other machine internal used to make tea. It delivers Tea after receiving trayEmpty from INIT and Deliver_Tea from MAC_INT.

A diagrammatic representation for VENDING_MACHINE2 is shown 5.5. As in VENDING_MACHINE1, dotted lines show optional actions which can take place at a particular time.

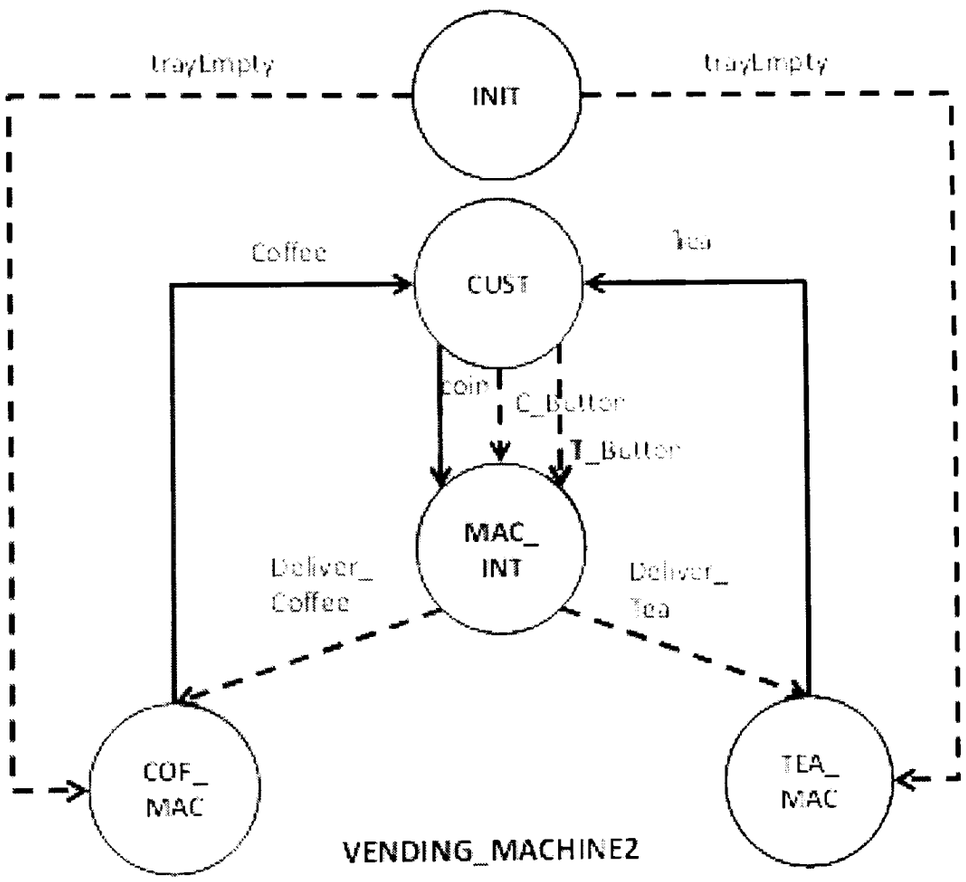


Figure 5.5: VENDING_MACHINE2.

Figure 5.6 shows the SACS specification for VENDING_MACHINE2.

```
//Init simply sends a trayEmpty once and terminate
INIT = trayEmpty!:@
//customer inserts a coin and presses either coffee button or tea button to
//receive a coffee or tea respectively
CUST =
coin!:(C_Button!+T_Button!):CUST + 1:(Coffee?+Tea?):@
// sends deliver coffee signal after receiving coin & C_Button or
//tea signal after receiving coin & T_Button
MAC_INT =
 (1:coin?:C_Button?:Deliver_Coffee! +
          1:coin?:T_Button?:Deliver_Tea!):MAC_INT
//coffee machine makes coffee after deliver coffee and trayEmpty signals
COF_MAC =
      (Deliver_Coffee?.trayEmpty?):Coffee!:COF_MAC
//tea machine makes tea after deliver tea and trayEmpty signals
TEA_MAC =  (Deliver_Tea?.trayEmpty?):Tea!:TEA_MAC
// Define the system
 SYSTEM = INIT x CUST x MACH_INT x COF_MAC x TEA_MAC
```

Figure 5.6: SACS specification for VENDING_MACHINE2.

The LTS used to express the SOS derived from the inference rules for VENDING_MACHINE2 is given below:

$$\frac{N_1 \quad N_2 \quad N_3 \quad N_4 \quad N_5}{INIT \,\mathrm{x}\, CUST \,\mathrm{x}\, MAC\_INT \,\mathrm{x}\, COF\_MAC \,\mathrm{x}\, TEA\_MAC}$$

The SOS specifications for the process nodes N_1, N_4, and N_5 are the same as that of the SOS specification for the VENDING_MACHINE1 namely N_1, N_5, and N_6 respectively.

Let N_2 denote CUST and its LTS transition is stated below:

$$N_2 = \frac{N_{2_1} \quad N_{2_2}}{\dfrac{CUST\_AGT1 + CUST\_AGT2 \xrightarrow{coffee? + tea?} CUST\_AGT1' + CUST\_AGT2'}{CUST \xrightarrow{coffee? + tea?} CUST'}} :: CUST$$

$$N_{2_1} = \frac{N_{2_{11}}}{CUST\_AGT1 \xrightarrow{0} CUST\_AGT1' : (coin! : (C\_Button! + T\_Button!))} :: CUST\_AGT1$$

or

$$N_{2_1} = \frac{N_{2_{12}}}{CUST\_AGT1 \xrightarrow{0} CUST\_AGT1' : (coin! : (C\_Button! + T\_Button!))} :: CUST\_AGT1$$

$$N_{2_{11}} = \frac{}{CUST\_AGT1 \xrightarrow{0} CUST\_AGT1' : coin! : C\_Button!} :: CUST\_AGT1$$

$$N_{2_{12}} = \frac{}{CUST\_AGT1 \xrightarrow{0} CUST\_AGT1' : coin! : T\_Button!} :: CUST\_AGT1$$

$$N_{2_2} = \frac{\overline{coffee? : CUST\_AGT2 \xrightarrow{coffee?} CUST\_AGT2' : @}}{(coffee? + tea?) : CUST\_AGT2 \xrightarrow{(coffee? + tea?)} CUST\_AGT2' : @} :: CUST\_AGT2$$

or

$$N_{2_2} = \frac{\overline{tea? : CUST\_AGT2 \xrightarrow{tea?} CUST\_AGT2' : @}}{(coffee? + tea?) : CUST\_AGT2 \xrightarrow{(coffee? + tea?)} CUST\_AGT2' : @} :: CUST\_AGT2$$

$$N_3 = \frac{N_{3_1}}{MAC\_INT \xrightarrow{(1:coin?:C\_Button? + 1:coin?:T\_Button?)} MAC\_INT' : (Deliver\_Coffee! + Deliver\_Tea!)} :: MAC\_INT$$

$$N_{3_1} = \frac{}{1 : coin? : C\_Button? : MAC\_INT \xrightarrow{1:coin?:C\_Button?} MAC\_INT' : Deliver\_Coffee!} :: MAC\_INT$$

OR

$$N_{3_1} = \frac{}{1 : coin? : T\_Button? : MAC\_INT \xrightarrow{1:coin?:T\_Button?} MAC\_INT' : Deliver\_Tea!} :: MAC\_INT$$

The two different implementations of the vending machine scenario are considered in order to study the equivalence relations between them. Equivalence relations are needed mainly for two purposes:

1. To prove that the specification meets the design of a system
2. To study the equality between two systems so that one system can be replaced with the other which may be simpler or cheaper.

The equivalence properties of process algebra are based on the operational semantics. The following section studies the equivalence properties of SACS.

5.3 Equivalence Relation Properties of SACS

Two expressions are said to be equivalent when no differences can be found between them and they both describe the same system.

Definition 2. Equivalence Relation
An equivalence relation between two processes $P, Q \in \Re$ can be written as $P \equiv Q$, a binary relation, such that it is :

- Reflexive: if $P \equiv P$
- Symmetrical: if $P \equiv Q$ then $Q \equiv P$
- Transitive: if $P \equiv Q$ and $Q \equiv S$ then $P \equiv S$.

Based on the structure of the transition relation, many different types of equivalences can be defined on processes. Most commonly studied process equivalences on LTS include Milner's simulation equivalence, bisimulation equivalence derived from simulation [Park, 1981], and trace equivalence etc.

In the rest of the section we discuss trace and bisimulation equivalences for SACS.

5.3.1 Trace Equivalence

$\cong_T$ A trace is sequence of observable actions when a process/agent moves from one state to another. Two processes are said to be trace equivalent if, and only if, they perform exactly the same sequence of observable actions. In other words, two processes are trace equivalent if and only if they engage in the same traces. A list of actions of an individual trace is shown by using enclosed brackets <list of actions>[Gray, 2000].
For example,
$< >$ denotes an empty trace. An empty trace describes the behaviour of an agent before it engages in its first action.
$< \alpha_1 >$ denotes a one action trace.
$< \alpha_1, \alpha_2 >$ denotes a two action trace in which α_1 is followed by α_2.

Definition 3. Trace Equivalence
A trace of a process P is a sequence $< \alpha_1, \alpha_2, ..., \alpha_n > \in \alpha^*(n \geq 0)$ such that there exists a sequence of transitions

$$P = P_0 \xrightarrow{\alpha_1} P_1 \xrightarrow{\alpha_2} ... \xrightarrow{\alpha_n} P_n$$

for some $P_0, P_1, ..., P_n$ where α^* is a set of sequences of traces. Let Q be another process. P and Q are trace equivalent if the traces of P are equivalent to traces of Q.

$$P \cong_T Q \Leftrightarrow (Traces(P) = Traces(Q))$$

Consider VENDING_MACHINE1 and VENDING_MACHINE2 explained in Section 5.2. Traces of both the examples when delivering coffee are given below:

$Traces\ of$ **VENDING_MACHINE1** $= (<>, < trayEmpty >, < coin, C\_Button >,$
$< coin, C\_Button, Deliver\_Coffee >, < coin, C\_Button, Deliver\_Coffee, Coffee >)$
$Traces\ of$ **VENDING_MACHINE2** $= (<>, < trayEmpty >, < coin, C\_Button >,$
$< coin, C\_Button, Deliver\_Coffee >, < coin, C\_Button, Deliver\_Coffee, Coffee >)$

Therefore, when delivering coffee,

$$Traces\ of\ \textbf{VENDING\_MACHINE1} = Traces\ of\ \textbf{VENDING\_MACHINE2}$$
$$\Rightarrow\ \textbf{VENDING\_MACHINE1} \cong_T \textbf{VENDING\_MACHINE2}$$

The point to be noted here is that $P \cong_T Q$ does not imply that the systems P and Q are the same. For example, consider the two systems P and Q shown in Figure 5.7.

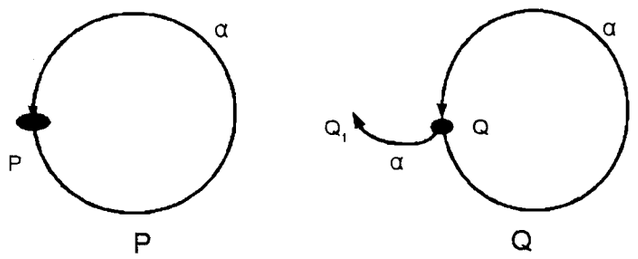


Figure 5.7: Example - Trace Equivalence

Traces of P are $\alpha^* = (<>, < \alpha >, < \alpha, \alpha >, < \alpha, \alpha, \alpha >, ..., < \alpha^n >)$ for some integer n.
Traces of Q are $\alpha^* = (<>, < \alpha >, < \alpha, \alpha >, < \alpha, \alpha, \alpha >, ..., < \alpha^n >)$ for some integer n.
Therefore, $P \cong_T Q$.

Trace equivalence uses traces to distinguish the behaviour of the system. It is not suitable when systems exhibit deadlock behaviour. An example may be the case of completed trace of a process P which has a trace sequence $< \alpha_1, \alpha_2, ..., \alpha_k > \in \alpha^* >$ where $k \geq 0$ such that the sequence transitions are specified as below:

$$P = P_0 \xrightarrow{\alpha_1} P_1 \xrightarrow{\alpha_2} ... \xrightarrow{\alpha_k} P_k \not\rightarrow$$

where $\nrightarrow$ denotes end of transitions for some $P_1, P_2, ..., P_k$. There cannot be further actions possible for P after P_k, and this is a clear case of deadlock behaviour.

The example shown in Figure 5.7 may seem to be equivalent according to the traces but when interactions between processes are considered, the machine can deadlock.

Instead of using traces to compare the system, actual behaviour can be used. Actual behaviour can be represented using actions and successor states. Studying the equivalence by comparing the actions and reachable states is called bisimulation. Bisimulation is based on the idea of observable behaviour and aims to capture the idea of equivalence as identical observed behaviour.

5.3.2 Bisimulation Equivalence

Definition 4. Simulation B
A binary relation $\mathbf{B} \subseteq \Re \mathrm{x} \Re$ is a simulation P **B** Q,
if $P \xrightarrow{\alpha_1} P'$ then there is $Q \xrightarrow{\alpha_1} Q'$ such that $P' \mathbf{B} Q'$ where $P, Q \in \Re$ and $\alpha_1 \in \alpha$.

Two processes are bisimulation equivalent if they are trace equivalent and the states that they reach are also equivalent. A relation is considered as an equivalent relation if it is:

- Reflexive, $P \mathbf{B} P$
- Symmetrical, $P \mathbf{B} Q \Rightarrow Q \mathbf{B} P$
- Transitive, $(P \mathbf{B} Q) \wedge (Q \mathbf{B} R) \Rightarrow P \mathbf{B} R$

where P, Q, $R \in \Re$ and **B** denotes bisimulation relation.

The study of bisimulation equivalence is based on the Labelled Transition Systems (LTS). This research focuses on two popular bisimulation equivalences for SACS: strong bisimulation and weak/observational bisimulation. While strong bisimulation compares both internal and external behaviours, the weak bisimulation compares only the external behaviours of the processes. This property is most widely used to study concurrent systems. If two systems are proved to be strongly bisimilar, they are also weakly bisimilar. The following sections discuss the bisimulation equivalences for SACS.

Strong Bisimulation Equivalence

Strong bisimulation checks whether two agents are equivalent in all their actions, both internal and external. The symbol $\sim$ is used to refer bisimulation.

Definition 5. Strong Bisimulation ($\sim$)
A binary relation over the set of states of an LTS is a strong bisimulation equivalence relation, $P \sim Q$ for $P, Q \in \Re$ and $\alpha_1 \in \alpha$:

$$\text{if } P \xrightarrow{\alpha_1} P' \text{ then for some } Q', \text{ if } Q \xrightarrow{\alpha_1} Q' \text{ such that } P' \mathbf{B} Q'$$

Conversely,

$$\text{if } Q \xrightarrow{\alpha_1} Q' \text{ then for some } P', \text{ if } P \xrightarrow{\alpha_1} P' \text{ such that } Q' \mathbf{B} P'$$

Example 6. [Gray, 2000]

Let P, Q, $Q_1 \in \Re$ and $\alpha_1 \in \alpha$ and
let $P \overset{def}{=\!=} \alpha_1 : P$ and $Q \overset{def}{=\!=} \alpha_1 : Q_1$, $Q_1 \overset{def}{=\!=} \alpha_1 : Q$ (shown in Figure 5.8).

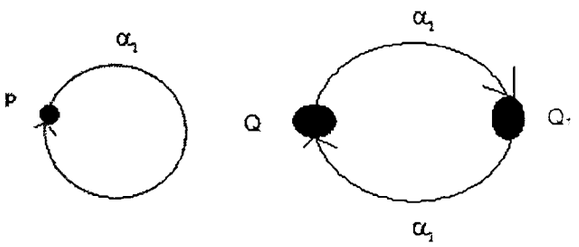


Figure 5.8: Example - Strong Bisimulation

Prove $P \sim Q$:
In the initial state, $P \sim Q$

$$if\ P \xrightarrow{\alpha_1} P \Rightarrow Q \xrightarrow{\alpha_1} Q_1\ and\ P \sim Q_1$$
$$if\ Q \xrightarrow{\alpha_1} Q_1 \Rightarrow P \xrightarrow{\alpha_1} P\ and\ P \sim Q_1$$

which can be written as:
Given $P \xrightarrow{\alpha_1} P \Rightarrow Q \xrightarrow{\alpha_1} Q_1$ and $Q \xrightarrow{\alpha_1} Q_1 \Rightarrow P \xrightarrow{\alpha_1} P$
then $P \sim Q$ if $P \sim Q_1$, and this can be expressed as $(P, Q) \subseteq \sim$ if $P \sim Q_1$.

Checking (P, Q_1):

$P \xrightarrow{\alpha_1} P \Rightarrow Q_1 \xrightarrow{\alpha_1} Q$ and $Q_1 \xrightarrow{\alpha_1} Q \Rightarrow P \xrightarrow{\alpha_1} P$
$\{(P, Q), (P, Q_1)\} \subseteq \sim$ if $P \sim Q$.
But (P, Q) is already in $\sim$. All the states of the processes simulate each other giving the strong bisimulation relation $\sim = \{(P, Q), (P, Q_1)\}$ which is complete and contains (P, Q) and therefore $P \sim Q$.

For all P, Q, R $\in \Re$ in SACS,

- P x Q $\sim$ Q x P
- P x 0 $\sim$ P and
- (P x Q) x R $\sim$ P x (Q x R)

Example 7. VENDING_MACHINE

Consider VENDING_MACHINE1 and VENDING_MACHINE2 explained in Section 5.2. To prove that $VENDING\_MACHINE1 \sim VENDING\_MACHINE2$, it must be proved that they simulate each other. We present the bisimulation on the LTS as these are the most common structures upon which bisimulation is studied. Having defined their SOS specifications for VENDING_MACHINE1 and VENDING_MACHINE2, the LTS have to be defined in order to be able to compare the transitions/actions. Let V_1 and V_2 denote VENDING_MACHINE1 and VENDING_MACHINE2 respectively. Let the LTS of $V_1 \stackrel{def}{=\!=} (S_1, K_1, T_1)$ where

$$S_1 \stackrel{def}{=\!=} \{INIT, CUST, MAC\_INT1, MAC\_INT2, COF\_MAC, TEA\_MAC\}$$
$$K_1 \stackrel{def}{=\!=} \{trayEmpty, coin, Deliver\_Coffee, Deliver\_Tea, Coffee, Tea\}$$

Based on the SOS specifications, the transitions for V_1 are derived and listed below:

$t1_1 \stackrel{def}{=\!=} INIT \xrightarrow{0} INIT' \cdot trayEmpty! : @$

$t1_2 \stackrel{def}{=\!=} CUST \xrightarrow{coffee? + tea?} CUST$

$t1_2$ can be expanded as

$t1_2 \stackrel{def}{=\!=} CUST\_AGT1 + CUST\_AGT2 \xrightarrow{coffee? + tea?} CUST\_AGT1' + CUST\_AGT2'$

The transition for $CUST\_AGT1$ is given below:

$t1_{2_1} \stackrel{def}{=\!=} CUST\_AGT1 \xrightarrow{0} CUST\_AGT1' (coin! : C\_Button! + coin! : T\_Button!)$

When '+' operator is applied in $(coin! : C\_Button! + coin! : T\_Button!)$, the resultant transitions would be either $t1_{2_{11}}$ or $t1_{2_{12}}$ depending upon the customer input. They are given below:

$t1_{2_{11}} \stackrel{def}{=\!=} CUST\_AGT1 \xrightarrow{0} CUST\_AGT1' : coin! : C\_Button!$

$t1_{2_{12}} \stackrel{def}{=\!=} CUST\_AGT1 \xrightarrow{0} CUST\_AGT1' : coin! : T\_Button!$

The internal transitions of $CUST\_AGT2$ are given below:

$t1_{2_2} \stackrel{def}{=\!=} (coffee? + tea?) : CUST\_AGT2 \xrightarrow{(coffee? + tea?)} CUST\_AGT2' : @$

When '+' operator is applied in $(coffee? + tea?)$, the resultant transitions would be either $t1_{2_{21}}$ or $t1_{2_{22}}$ depending upon the customer input. They are stated below:

$t1_{2_{21}} \stackrel{def}{=\!=} Coffee? : CUST\_AGT2 \xrightarrow{Coffee?} CUST\_AGT2' : @$

$t1_{2_{22}} \stackrel{def}{=\!=} Tea? : CUST\_AGT2 \xrightarrow{Tea?} CUST\_AGT2' : @$

$t1_3 \stackrel{def}{=\!=} 1 : coin? : C\_Button? : MAC\_INT1 \xrightarrow{1 : coin? : C\_Button?} MAC\_INT1' : Deliver\_Coffee!$

$t1_4 \stackrel{def}{=\!=} 1 : coin? : T\_Button : MAC\_INT2 \xrightarrow{1 : coin? : T\_Button?} MAC\_INT2' : Deliver\_Tea!$

$t1_5 \stackrel{def}{=\!=} Deliver\_Coffee? : trayEmpty? : COF\_MAC$

$$\xrightarrow{Deliver\_Coffee? : trayEmpty?} Coffee! : COF\_MAC'$$

$t1_6 \stackrel{def}{=\!=} Deliver\_Tea? : trayEmpty? : TEA\_MAC$

$$\xrightarrow{Deliver\_Tea? : trayEmpty?} Tea! : TEA\_MAC'$$

Including all the internal transitions,

$$T_1 \stackrel{def}{=\!=} \{t1_1, t1_{2_{11}}, t1_{2_{12}}, t1_{2_{21}}, t1_{2_{22}}, t1_3, t1_4, t1_5, t1_6\}$$

Let the LTS of $V_2 \stackrel{def}{=\!=} (S_2, K_2, T_2)$ where

$$S_2 \stackrel{def}{=\!=} \{INIT, CUST, MAC\_INT, COF\_MAC, TEA\_MAC\}$$
$$K_2 \stackrel{def}{=\!=} \{trayEmpty, coin, Deliver\_Coffee, Deliver\_Tea, Coffee, Tea\}$$

Based on the SOS specifications, the transitions for V_2 are derived and listed below:

$t2_1 \stackrel{def}{=\!=} INIT \xrightarrow{0} INIT' \cdot trayEmpty! : @$

$t2_2 \stackrel{def}{=\!=} CUST \xrightarrow{coffee? + tea?} CUST$

$t2_2$ can be expanded as

$t2_2 \stackrel{def}{=\!=} CUST\_AGT1 + CUST\_AGT2 \xrightarrow{coffee? + tea?} CUST\_AGT1' + CUST\_AGT2'$

The transition for $CUST\_AGT1$ is given below:

$t2_{2_1} \stackrel{def}{=\!=} CUST\_AGT1 \xrightarrow{0} CUST\_AGT1' (coin! : (C\_Button! + T\_Button!))$

The ':' operator is a sequential AND operator and is distributive. So,

$(coin! : (C\_Button! + T\_Button!)) = (coin! : C\_Button! + coin! : T\_Button!)$.

Based on this, the resultant transitions would be either $t1_{2_{11}}$ or $t1_{2_{12}}$ of V_1 depending upon the customer input.

$t2_{2_{11}} \stackrel{def}{=\!=} CUST\_AGT1 \xrightarrow{0} CUST\_AGT1' : coin! : C\_Button!$

$t2_{2_{12}} \stackrel{def}{=\!=} CUST\_AGT1 \xrightarrow{0} CUST\_AGT1' : coin! : T\_Button!$

The internal transitions of $CUST\_AGT2$ are given below:

$t2_{2_2} \stackrel{def}{=\!=} (coffee? + tea?) : CUST\_AGT2 \xrightarrow{(coffee? + tea?)} CUST\_AGT2' : @$

When '+' operator is applied in $(coffee? + tea?)$, the resultant transitions would be either $t1_{2_{21}}$ or $t2_{2_{22}}$ depending upon the customer input. They are stated below:

$t2_{2_{21}} \stackrel{def}{=\!=} Coffee? : CUST\_AGT2 \xrightarrow{Coffee?} CUST\_AGT2' : @$

$t2_{2_{22}} \stackrel{def}{=\!=} Tea? : CUST\_AGT2 \xrightarrow{Tea?} CUST\_AGT2' : @$

$t2_3 \stackrel{def}{=\!=} (1 : coin? : C\_Button? + 1 : coin? : T\_Button?) : MAC\_INT$

$$\xrightarrow{(1:coin?:C\_Button? + 1:coin?:T\_Button?)} MAC\_INT' : (Deliver\_Coffee! + Deliver\_Tea!)$$

When '+' operator is applied in $(1 : coin? : C\_Button? + 1 : coin? : T\_Button?))$, the resultant transitions would use either $(1 : coin? : C\_Button?)$ or $(1 : coin? : T\_Button?)$ depending upon the customer input. The resultant transitions are given below:

$t2_{3_1} \stackrel{def}{=\!=} 1 : coin? : C\_Button? : MAC\_INT$

$$\xrightarrow{1:coin?:C\_Button?} MAC\_INT' : DeliverCoffee!$$

$t2_{3_2} \stackrel{def}{=\!=} 1 : coin? : T\_Button? : MAC\_INT$

$$\xrightarrow{1:coin?:T\_Button?} MAC\_INT' : DeliverTea!$$

$t2_4 \stackrel{def}{=\!=} DeliverCoffee? : trayEmpty? : COF\_MAC$

$$\xrightarrow{DeliverCoffee?:trayEmpty?} Coffee! : COF\_MAC'$$

$t2_5 \stackrel{def}{=\!=} DeliverTea? : trayEmpty? : TEA\_MAC$

$$\xrightarrow{DeliverTea?:trayEmpty?} Tea! : TEA\_MAC'$$

Including all the internal transitions,

$$T_2 \stackrel{def}{=\!=} \{t2_1, t2_{2_{11}}, t2_{2_{12}} t1_{2_{21}}, t1_{2_{22}}, t2_{3_1}, t2_{3_2}, t2_4, t2_5\}$$

Proof: To prove $V_1 \sim V_2$,

Checking (INIT of V_1, INIT of V_2),

$$t1_1 \text{ in } V_1 \Rightarrow t2_1 \text{ in } V_2 \text{ and } t2_1 \text{ in } V_2 \Rightarrow t1_1 \text{ in } V_1$$
$$\{(INIT \text{ of } V_1,\ INIT \text{ of } V_2)\} \subseteq \sim \text{ if } INIT' \text{ of } V_1 \sim INIT' \text{ of } V_2$$

Checking (CUST of V_1, CUST of V_2),

Both CUST of V_1 and CUST of V_2 consist of CUST_AGT1 and CUST_AGT2. When their transitions are considered,

$$t1_{2_{11}}, t1_{2_{12}}, t1_{2_{21}}, t1_{2_{22}} \text{ in } V_1 \Rightarrow t2_{2_{11}}, t2_{2_{12}} t1_{2_{21}}, t1_{2_{22}} \text{ in } V_2$$
$$t2_{2_{11}}, t2_{2_{12}} t1_{2_{21}}, t1_{2_{22}} \text{ in } V_2 \Rightarrow tt1_{2_{11}}, t1_{2_{12}}, t1_{2_{21}}, t1_{2_{22}} \text{ in } V_1$$

$\{(CUST\_AGT1 \text{ of } V_1,\ CUST\_AGT1 \text{ of } V_2)\} \subseteq \sim$

$$\text{if } CUST\_AGT1' \text{ of } V_1 \sim CUST\_AGT1' \text{ of } V_2$$

$\{(CUST\_AGT2 \text{ of } V_1,\ CUST\_AGT2 \text{ of } V_2)\} \subseteq \sim$

$$\text{if } CUST\_AGT2' \text{ of } V_1 \sim CUST\_AGT2' \text{ of } V_2$$

$\Rightarrow \{(CUST \text{ of } V_1,\ CUST \text{ of } V_2)\} \subseteq \sim \text{ if } CUST' \text{ of } V_1 \sim CUST' \text{ of } V_2$ Therefore, $\{($ INIT of V_1, INIT of $V_2)$, (CUST of V_1, CUST of $V_2)\} \subseteq \sim$

Checking (MAC_INT1 of V_1, MAC_INT of V_2),

$$t1_3 \text{ in } V_1 \Rightarrow t2_{3_1} \text{ in } V_2 \text{ and } t2_{3_1} \text{ in } V_2 \Rightarrow t1_3 \text{ in } V_1$$

$\{(MAC\_INT1 \text{ of } V_1,\ MAC\_INT \text{ of } V_2)\} \subseteq \sim$

$$\text{if } MAC\_INT1' \text{ of } V_1 \sim MAC\_INT' \text{ of } V_2$$

Checking (MAC_INT2 of V_1, MAC_INT of V_2),

$$t1_4 \; in \; V_1 \Rightarrow t2_{3_2} \; in \; V_2 \text{ and } t2_{3_2} \; in \; V_2 \Rightarrow t1_4 \; in \; V_1$$

$\{(MAC\_INT2 \; of \; V_1, \; MAC\_INT \; of \; V_2)\} \subseteq \sim$

$$if \; MAC\_INT2' \; of \; V_1 \sim MAC\_INT' \; of \; V_2$$

Therefore,

$$\{(\text{ INIT of } V_1, \text{ INIT of } V_2), (\text{CUST of } V_1, \text{ CUST of } V_2),$$
$$(\text{MAC\_INT1 of } V_1, \text{ MAC\_INT of } V_2), (\text{MAC\_INT2 of } V_1, \text{ MAC\_INT of } V_2)\} \subseteq \sim$$

Checking (COF_MAC of V_1, COF_MAC of V_2),

$$t1_5 \; in \; V_1 \Rightarrow t2_4 \; in \; V_2 \text{ and } t2_4 \; in \; V_2 \Rightarrow t1_5 \; in \; V_1$$

$\{(\text{COF\_MAC of } V_1, \text{ COF\_MAC of } V_2)\} \subseteq \sim$

$$\text{if COF\_MAC' of } V_1 \sim \text{COF\_MAC' of } V_2$$

Therefore,

$$\{(\text{ INIT of } V_1, \text{ INIT of } V_2), (\text{CUST of } V_1, \text{ CUST of } V_2),$$
$$(\text{MAC\_INT1 of } V_1, \text{ MAC\_INT of } V_2), (\text{MAC\_INT2 of } V_1, \text{ MAC\_INT of } V_2),$$
$$(\text{COF\_MAC of } V_1, \text{ COF\_MAC of } V_2)\} \subseteq \sim$$

Checking (TEA_MAC of V_1, TEA_MAC of V_2),

$$t1_6 \; in \; V_1 \Rightarrow t2_5 \; in \; V_2 \text{ and } t2_5 \; in \; V_2 \Rightarrow t1_6 \; in \; V_1$$

$\{(TEA\_MAC \text{ of } V_1, TEA\_MAC \text{ of } V_2)\} \subseteq \sim$

$$\text{if } TEA\_MAC' \text{ of } V_1 \sim \text{TEA\_MAC' of } V_2$$

Therefore,

$$\{(\textbf{ INIT of } V_1, \textbf{ INIT of } V_2), (\textbf{CUST of } V_1, \textbf{ CUST of } V_2),$$
$$(\textbf{MAC\_INT1 of } V_1, \textbf{ MAC\_INT of } V_2), (\textbf{MAC\_INT2 of } V_1, \textbf{ MAC\_INT of } V_2),$$
$$(\textbf{COF\_MAC of } V_1, \textbf{ COF\_MAC of } V_2), (\textbf{TEA\_MAC of } V_1, \textbf{ TEA\_MAC of } V_2)\} \subseteq \sim$$

Every action in each state of each process of V_1 can be simulated by an action of V_2 and hence, it can be concluded that,

$$V_1 \sim V_2$$

Strong bisimulation satisfies many of the properties of trace equivalence and congruence which is discussed later in this section. A relation which is behavioural equivalent is not necessarily strongly bisimilar. For example, $\alpha_1 : \tau : 0$ and $\alpha_1 : 0$ are behaviourally equivalent but are not strongly bisimilar as their internal actions differ.

Weak/observable Bisimulation Equivalence

There are systems in which external actions are compared, in other words, the black box behaviours of the systems are matched. Such an equivalence in which only the observable external actions are compared is called weak bisimulation or observational equivalence. The symbol $\approx$ is used to denote observational equivalence, and is written as $P \approx Q$ for $P, Q \in \Re$. Asynchronous communication can easily satisfy observational equivalence.

Definition 6. Observational Equivalence ($\approx$)
A binary relation over a set of states of the LTS system is an observational equivalence relation:
if $P \xrightarrow{\alpha_1} P'$ then for some Q', $Q \xrightarrow{\alpha_1} Q'$ and $P' \approx Q'$
Conversely,
if $Q \xrightarrow{\alpha_1} Q'$ then for some P', $P \xrightarrow{\alpha_1} P'$ and $P' \approx Q'$
where $P, Q \in \Re$ and $\alpha_1 \in \alpha$

In such a relation, $1 : \alpha_1 : 0$ and $\alpha_1 : 0$ are equivalent. This is due to the fact that the weak/observable bisimulation ignores internal actions. Consider an example consisting of two processes: $1 : \alpha_1 : 0 + \alpha_2 : 0$ and $\alpha_1 : 0 + \alpha_2 : 0$ as shown in Figure 5.9.

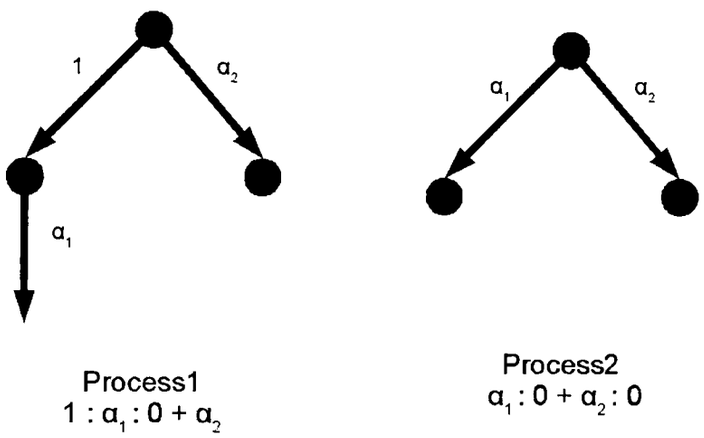


Figure 5.9: $1 : \alpha_1 : 0 + \alpha_2 : 0$ and $\alpha_1 : 0 + \alpha_2 : 0$

Observational equivalence considers these two processes to be equivalent, but in real life they behave differently. The second process may perform α_1 or α_2, but if the first one engage in an internal action, it will only perform α_1. According to Gray (2000):
$P \approx Q$ if, for every action of P, visible or invisible, $P \xrightarrow{\alpha_1} P', Q$ can engage in an α_1 action surrounded by any number, n, of internal actions where n can be a 0, and move to state Q' such that $P' \approx Q'$. The same holds if P and Q are interchanged.
In order to clarify, $\Rightarrow$ is introduced instead of $\rightarrow$. The notation $\stackrel{\alpha_1}{\Rightarrow}$ denotes that a transition composed of an α_1 action is surrounded by any number including zero of internal actions.

Definition 7. ($\stackrel{\alpha_1}{\Rightarrow}$)
$P \stackrel{\alpha_1}{\Rightarrow} P'$ is equivalent to $P \xrightarrow{1^*} \xrightarrow{\alpha_1} \xrightarrow{1^*} P'$ where 1^* represents zero or more '1' actions.

Example
For $Q \stackrel{def}{=\!=} 1 : \alpha_1 : Q'$ following transitions are valid and true.

$Q \stackrel{1}{\Rightarrow} \alpha : 1 : Q'$
$Q \stackrel{\alpha_1}{\Rightarrow} \alpha : Q'$
$Q \stackrel{\alpha_1}{\Rightarrow} Q'$

Therefore, processes can change the state by engaging in actions that are internal or external. Considering this factor, a modified definition for observational equivalence is stated below.

Definition 8. (Modified Observational Equivalence)
A binary relation over a set of states of the LTS system is a weak bisimulation if and only if whenever $P \approx Q$:

if $P \xrightarrow{\alpha_1} P'$, then there is a transition $Q \stackrel{\alpha_1}{\Rightarrow} Q'$ and $P' \approx Q'$
if $Q \xrightarrow{\alpha_1} Q'$, then there is a transition $P \stackrel{\alpha_1}{\Rightarrow} P'$ and $Q' \approx P'$.
where $P, Q \in \Re$ and $\alpha_1 \in \alpha$

From the above definitions, weak/observational bisimulation equivalence relation can be redefined as below.

Definition 9. (Weak/Observational Bisimulation Equivalence)
A binary relation over a set of states of the LTS system is a weak bisimulation, $P \approx Q$:

- if $P \xrightarrow{\alpha_1} P'$, then either
  - $\alpha_1 = 1$ and $P' \approx Q$ (or)
  - for some Q', $Q \stackrel{\alpha_1}{\Rightarrow} Q'$ and $P' \approx Q'$

  and conversely,
- if $Q \xrightarrow{\alpha_1} Q'$ then either
  - $\alpha_1 = 1$ and $P \approx Q'$ (or)
  - for some P', $P \stackrel{\alpha_1}{\Rightarrow} P'$ and $P' \approx Q'$

where $P, Q \in \Re$ and $\alpha_1 \in \alpha$.

Example: [Gray, 2000]
Let $P \in \Re$, $1 \in \tau$ and $\alpha_1 \in \alpha$, Prove $P = 1 : P$.

Initial actions of P are external and denoted as $P = \alpha_1 : Q$, and

Initial actions of P are internal and denoted as $P = 1 : Q$.
To find $P = 1 : P$,
1. **Checking** $(\alpha_1 : Q, 1 : \alpha_1 : Q)$

$$\alpha_1 : Q \xrightarrow{\alpha_1} Q_1 \Rightarrow 1 : \alpha_1 : Q \overset{\alpha_1}{\Rightarrow} Q \quad and \quad Q \approx Q$$
$$1 : \alpha_1 : Q \xrightarrow{1} : \alpha_1 : Q \quad as \quad \alpha_1 = 1 \quad we\ need\ \alpha_1 : Q = \alpha_1 : Q$$

2. **Checking** $(1 : Q, 1 : 1 : Q)$

$$1 : Q \xrightarrow{1} Q \Rightarrow 1 : 1 : Q' \Rightarrow Q \quad and \quad Q \approx Q$$
$$1 : 1 : Q \xrightarrow{1} 1 : Q \quad as \quad \alpha_1 = 1 \quad we\ need\ 1 : Q \approx 1 : Q$$

Congruence ($\cong$)

For a process equivalence to be practically useful, it must be congruent. Two systems are said to be congruent, when no observation can distinguish between them and two (sub)systems are said to be congruent if the result of placing them in the same system context yields two equivalent systems [Gray, 2000]. This means that two behaviourally equivalent processes can be used interchangeably as part of a large process without affecting the overall behaviour. This is crucial in inductive reasoning. For example, in specification and verification of software, replacing a subsystem with an equivalent one should ensure that the behaviour of the entire system is equivalent to the original system. The definition for congruence derived from CCS is stated below:

Definition 10. (SACS Congruence)
An equivalence relation $\cong \subseteq \Re$ is said to be congruence if it is preserved by the SACS constructs. In other words,
if $P \cong Q$ where $P, Q \in \Re$ then,

$$a : P : b \cong a : Q : b$$
$$P \text{ x } R \cong Q \text{ x } R$$
$$R \text{ x } P \cong R \text{ x } Q$$
$$new\ k\, P \cong new\ k\, Q$$

for every $a \in \alpha$, $b \in \beta$, $R \in \Re$, *and* $k \in \kappa$.

The SOS for SACS is extended with a further rule to include congruence and it is stated as follows:
If $P \cong P'$ and $Q \cong Q'$, then

$$\frac{P \cong P' \xrightarrow{\alpha} Q' \cong Q}{P \xrightarrow{\alpha} Q} :: CONGRUENT.$$

5.4 An Equivalence Relation for the SACS and AMPS

SACS has been used for the high level specification of the communication part of LIPS programs and is implemented using the Asynchronous Message Passing Systems (AMPS). It is necessary to study the proof of equivalence of SACS and AMPS to prove the completeness of AMPS. The semantics of both SACS and AMPS have been defined using Structural Operational Semantics (SOS) in terms of Labelled Transition Systems. We have two labelled transition system semantics: one for SACS and one for AMPS. If we can show the bisimulation equivalence between these two labelled transition systems then we can say that SACS and AMPS are equivalent.

We consider the weak bisimulation equivalence between SACS and AMPS. The reason is that when we compile a language or specification to another, it is very unlikely that we can faithfully preserve the operational semantics. This means that a transition from $P' \xrightarrow{\alpha} Q'$ in SACS may become a sequence of transitions in AMPS, namely $P' \xrightarrow{\alpha 1} \ldots \xrightarrow{\alpha n} Q'$ where most of $\alpha 1, \ldots, \xrightarrow{\alpha n}$. It might also be that we do not reach Q but a process equivalent to Q.

Figure 5.10 summarises the main results of the proof of equivalence between SACS and AMPS.

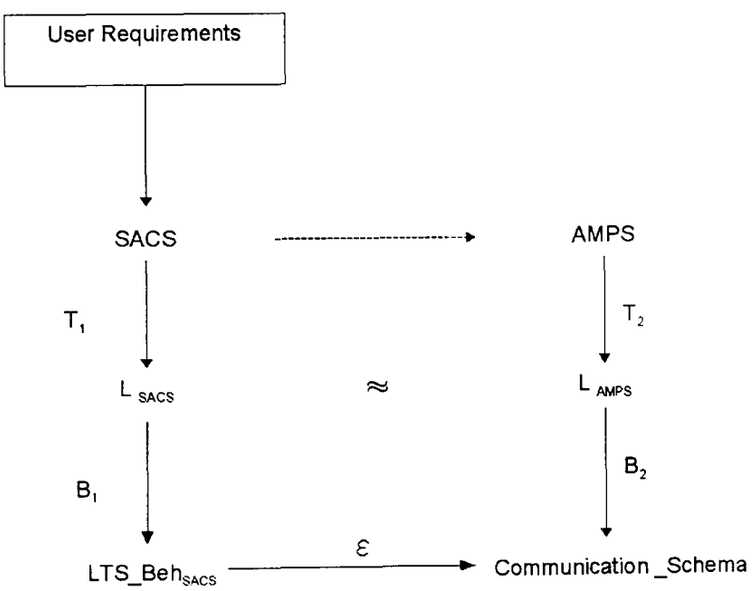


Figure 5.10: Summary of the proof of equivalence between SACS and AMPS

Given the user requirements, let L_{SACS} denote the Structured Operational Semantics defined using the Labelled Transition System (LTS) for SACS and the function T_1 mapping

the SACS to its LTS is shown below:

$$T_1 : SACS \rightarrow L_{SACS}$$

The behaviour of SACS has been defined using the sequences of LTS configurations which are used to define the SOS for SACS. It can be represented using a mapping B_1 from LTS, L_{SACS}, to its set of behaviours, LTS_Beh_{SACS}, which is shown below:

$$B_1 : L_{SACS} \rightarrow LTS\_Beh_{SACS}$$

The communication part of LIPS program written using its SACS specification is implemented using the AMPS. The relationship between the SACS and AMPS is shown using dotted lines in Figure 5.10. The SOS for AMPS is the set of LTS configurations denoted by L_{AMPS}. The function T_2 mapping the AMPS to its LTS is shown below:

$$T_2 : AMPS \rightarrow L_{AMPS}$$

The behaviour of AMPS is defined using its communication schema implemented using a set of functions. The mapping, B_2, from AMPS to the communication schema is shown below:

$$B_2 : L_{AMPS} \rightarrow Communication\_Schema$$

The behaviour of AMPS depends on the communication schema described in Sections 3.3 and 3.4 of Chapter 3. When the specifications of SACS have to be implemented, the SACS transitions have to be extended to include the communication schema of the AMPS. The extension function, ε, to be included to the SACS behaviour for its implementation purposes is shown below:

$$\varepsilon : LTS\_Beh_{SACS} \rightarrow Communication\_Schema$$

We can say that the set of configurations of L_{AMPS} is the set of configurations of L_{SACS} and a set of functions used to implement asynchronous message passing.

Set of configurations of SACS is given as:

$$L_{SACS} = \{Guard, GuardedProcess, Node, ConcurrencyComposition, System\}$$

Set of configurations of AMPS is given as:

$$L_{AMPS} = L_{SACS} \cup \{Is\_input\_available,\ Is\_ok\_to\_send,\ Send\}$$

An equivalence relation has to be constructed between the configurations, L_{SACS} and L_{AMPS}. This can be expressed using the behaviour of their configurations which can be

shown as below:

$$Communication\_Schema \approx \varepsilon \cup LTS\_Beh_{SACS}$$

where ε is the set of functions, *Is_input_available*, *Is_ok_to_send*, *Send*.

Definition 11. (Bisimilarity between two Labelled Transition Systems)
Two equally labelled transition systems L_1 and L_2 are bisimilar (written as $L_1 \approx L_2$) if and only if $L_i \stackrel{def}{=\!=} (S_i, K, T_i)$ for $i = 1, 2$ and there exists a relation $R \subseteq S_1 \text{ x } S_2$ such that $k \in K$:

1. $p \in S_1 \Rightarrow \exists q \,.\, q \in S_2 \wedge p, q \in R \text{ and } q \in S_2 \Rightarrow \exists p \,.\, p \in S_1 \wedge p, q \in R$
2. $\forall p\ q\ p' \,.\, p, q \in R \wedge p \rightarrow p' \Rightarrow \exists q' \,.\, q \rightarrow q' \wedge p', q' \in R$
3. $\forall p\ q\ q' \,.\, p, q \in R \wedge q \rightarrow q' \Rightarrow \exists p' \,.\, p \rightarrow p' \wedge p', q' \in R$

We show the equivalence relation between SACS and AMPS using Theorem 1.

Theorem 1. Theorem: Equivalence Relation for the SACS and AMPS
For every correctly labelled AMPS specification,

$$L_{AMPS} \approx L_{SACS}$$

Proof: The first step in the proof is to identify related configurations for
$(S_1, k, t), (S_2, k, t) \in R$ and

$(S_2, k, t) = (S_1, k, t) \cup set\ of\ functions.$

Then bisimilarity is proved using the three conditions stated in Definition 11.

Condition 1: Every pair $(S_1, k, t), (S_2, k, t) \in S_1 \text{ x } S_2$ must be in R. That is we must show that $(S_2, k, t) = (S_1, k, t) \cup set\ of\ functions$:

Let 'G' is a guard, 'GP' is a guarded process, 'N' is a node, 'S' is a set of parallel nodes, 'f_1, f_2, f_3' are the functions *Is_input_available*, *Is_ok_to_send*, and *Send* respectively.
$(S_1, k, t) = (\{G, GP, N, S\}, k, t)$

$$\begin{aligned}(S_2, k, t) &= (\{G, GP, N, S, f_1, f_2, f_3\}, k, t)\\ &= (S_1 \cup set\ of\ functions, k, t)\\ &\quad \text{where set of functions} = \{f_1, f_2, f_3\}\\ &= (S_1, k, t) \cup set\ of\ functions\end{aligned}$$

Condition 2 and condition 3 state that from any pair $(S_1, k, t), (S_2, k, t) \in R$ every L_{SACS} transition to (S_1', k', t') must have a corresponding L_{AMPS} transition and every

L_{AMPS} transition to (S_2', k', t') must have a corresponding L_{SACS} transition. Also, the resulting pair should satisfy the condition,

$$((S_1, k, t), (S_2, k, t)) \in R.$$

For each configuration of AMPS and SACS,

1. Find the $(S_1\prime, k\prime, t\prime)$ reached by α transition.
2. Assign $(S_2, k, t) = (S_1, \cup \; set\; of\; functions\, k, t)$ so that the configurations are in R.
3. Find the $(S_2\prime, k\prime, t\prime)$ reached by the α transition.
4. Check $((S_1\prime, k\prime, t\prime), (S_2\prime, k\prime, t\prime)) \in R$. This is done by checking that

$$Communication\_Schema \approx \varepsilon \cup LTS\_Beh_{SACS}$$

for each of the configurations of SACS and AMPS and they are listed below:

1. **Guard:**
   The L_{SACS} and L_{AMPS} configurations for a guard are shown below:
   L_{SACS} : $(\alpha_1 \circ \alpha_2 \circ \cdots \circ \alpha_k, s_1) \xrightarrow{\underline{T}} (\underline{T}\{\alpha_1, \alpha_2, \cdots \alpha_k\}, s_2)$
   L_{AMPS} : $((fch_{i1} \wedge fch_{i2} \wedge \cdots \wedge fch_{im}), s_1) \xrightarrow{\underline{T}} (\underline{T}\{ch_{i1}, ch_{i2}, \cdots, ch_{im}\}, s_2)$
   The function *Is_input_available* is called when an input channel in a guard checks for the availability of a new value. If a new value is available, the function will return 1. The configuration for the function is given as:
   $(Is\_input\_available(m, n), s_1) \xrightarrow{m,n} (1, s_1')$ where m, n specify the node number and variable number respectively.
   When this function returns 1, the *Send* function will be called to send the data from the DS of the AMPS to the respective channel. The configuration for *Send* function is given as:
   $(Send(m, n, t, d), s_1) \xrightarrow{m,n,t,d} (1, s_1')$.
   These two functions will be called consecutively for all the input channels and they are inter transitions. As a result, the system will move from state s_1 to s_2. The equivalence relation checks only the initial and final states and they are one and the same as that of the L_{SACS}.
   $\Rightarrow$ $Communication\_Schema \approx \varepsilon \cup LTS\_Beh_{SACS}$

Therefore,

$(\alpha_1 \circ \alpha_2 \circ \cdots \circ \alpha_k,\ s_1) \xrightarrow{\underline{T}} (\underline{T}\{\alpha_1, \alpha_2, \cdots \alpha_k\},\ s_2)$

$\approx\ ((fch_{i1} \wedge fch_{i2} \wedge \cdots \wedge fch_{im}),\ s_1) \xrightarrow{\underline{T}} (\underline{T}\{ch_{i1},\ ch_{i2},\ \cdots, ch_{im}\},\ s_2)$

2. **Guarded Process:**

The L_{SACS} and L_{AMPS} configurations for a guarded process are shown below:

L_{SACS}: $(\eta\ \texttt{Proc},\ s_1) \xrightarrow{\eta} (\texttt{Proc},\ s_2\{\sigma\})$

where η is a guard which becomes true to execute the associated process *proc* and generate values of the set of output characters σ.

L_{AMPS}: $(if\ G_i\ then\ P_i,\ s_1) \xrightarrow{CH} (OCH_i,\ s_2)$

When G_i becomes true, the associated process body is executed to generated 0 or more values for the output channels OCH_i. For every value available in the output channel, *Is_ok_to_send* function is called to find the value that can be sent to the data structure of AMPS.

$(Is\_ok\_to\_send(m,\ n),\ s_1) \xrightarrow{m,n} (1,\ s_1')$ where m, n specify the node number and variable number respectively.

If it is ok to send, the *Send* function is called to send the value to the DS.

$\Rightarrow$ $Communication\_Schema \approx \varepsilon \cup LTS\_Beh_{SACS}$

Therefore, $(\eta\ \texttt{Proc},\ s_1) \xrightarrow{\eta} (\texttt{Proc},\ s_2\{\sigma\}) \approx (if\ G_i\ then\ P_i,\ s_1) \xrightarrow{CH} (OCH_i,\ s_2)$

3. **Node:**

The L_{SACS} and L_{AMPS} configurations for a node are shown below:

L_{SACS}: $(\sum_{i=1}^{k} \eta_i\ \texttt{Proc}_i,\ s_1) \xrightarrow{for\ any\ i=1\ to\ k,\eta_i} (\texttt{Proc}_i,\ s_2\{\sigma_i\})$

where $\eta_i\ \texttt{Proc}_i$ is a guarded process GP_i. In other words, $\Re = GP_1 + GP_2 + ... + GP_k$ where $\Re$ is a node and GP_1, GP_2, ..., GP_k are guarded processes.

L_{AMPS}: $(while\ (\underline{T})\ do\ (GP_1\ else\ GP_2\ else\ ...\ else\ GP_k),\ s_1) \xrightarrow{\underline{T}}$

$((if\ (\underline{T})\ then\ GP_i;\ while\ (\underline{T})\ do\ (GP_1\ else\ GP_2\ else\ ...\ else\ GP_k)),\ s_2\{OCH_i\})$

The AMPS configuration shows the actual implementations of the SACS and expresses the guarded processes involved in making a node. In actual implementation, this is an infinite *while loop*. For a guarded process to get executed its guard should become true. When a guard is true, its associated process body is executed to generate values for 0 or more output channels.

$\Rightarrow$ $Communication\_Schema \approx \varepsilon \cup LTS\_Beh_{SACS}$

Therefore, $(\sum_{l=1}^{k} \eta_i\ \texttt{Proc}_i,\ s_1) \xrightarrow{for\ any\ i=1\ to\ k,\eta_i} (\texttt{Proc}_i,\ s_2\{\sigma_i\}) \approx$

while $(\underline{T})$ *do* $(GP_1$ *else* GP_2 *else ... else* $GP_k)$, $s_1) \xrightarrow{\underline{T}}$
$((if\ (\underline{T})\ then\ GP_i;\ while\ (\underline{T})\ do\ (GP_1\ else\ GP_2\ else\ ...\ else\ GP_k)), s_2\{OCH_i\})$

4. **System/Network:**
   The L_{SACS} and L_{AMPS} configurations for a node are shown below:

   Let there be n nodes in a network,

   L_{SACS}: $(\Re_1\ \mathbf{x}\ \Re_2\ \mathbf{x}\ ...\ \mathbf{x}\Re_m\ , s_1) \xrightarrow{\mu} (\Re'_1\ \mathbf{x}\ \Re'_2\ \mathbf{x}\ ...\ \mathbf{x}\ \Re'_m\ , s_2)$

   where $\Re_1$, $\Re_2$, ... , $\Re_n$ denote the nodes and μ is a prefix.
   If $\Re_1$ and $\Re_2$ are two nodes to be executed concurrently, their concurrency composition is denoted by $\Re_1\ \mathbf{x}\ \Re_2$. Once the nodes are concurrently executed, the system changes its state from s_1 to s_2.
   L_{AMPS}: The network definition part of a LIPS program (Refer Section 3.1 in Chapter 3) is defined using a set of connect statements. The configuration for a set of connect statements is given below:
   $\forall i : 1 \leq i \leq n((R_i(ich_{i1} \wedge ich_{i2} \wedge ... \wedge ich_{im}), s_1) \longrightarrow ((och_{i1}, och_{i2}, ..., och_{is}), s_2))$
   Based on these connect statements, the DS and DM of AMPS is created and initialised. The connect statements are implemented using the nodes definition. When these nodes execute, they make use of the set of input channels and produce values for the set of output channels using guarded processes. Once all the nodes finish executing to produce the intended output, the system will change its state from s_1 to s_2.
   $\Rightarrow$ $Communication\_Schema \approx \varepsilon \cup LTS\_Beh_{SACS}$
   Therefore, $(\Re_1\ \mathbf{x}\ \Re_2\ \mathbf{x}\ ...\ \mathbf{x}\Re_m\ , s_1) \xrightarrow{\mu} (\Re'_1\ \mathbf{x}\ \Re'_2\ \mathbf{x}\ ...\ \mathbf{x}\ \Re'_m\ , s_2) \approx$
   $\forall i : 1 \leq i \leq n((R_i(ich_{i1} \wedge ich_{i2} \wedge ... \wedge ich_{im}), s_1) \longrightarrow ((och_{i1}, och_{i2}, ..., och_{is}), s_2))$

Thus for every (S_1, k, t) in R there exists a (S_2, k, t) in R as required.

□

5.5 Summary

This chapter describes the formalisms supporting the communication part of LIPS using SACS through examples. The four rules used in SACS form a framework for successful concurrent program design in the context of point-to-point intercommunication patterns.

As SACS is already existing, this work has

1. defined the Structural Operational Semantics (SOS) using Labelled Transition Systems (LTS), and

2. studied the various equivalence properties of SACS.

Studying the equivalence properties is necessary to prove that the completed design meets its specification. It is also used to study the equality between two designs so that one design can be replaced by an equivalent if it is simpler or cheaper.

SACS is a formal specification tool to specify the asynchronous communication in a LIPS program. This is implemented using the Asynchronous Message Passing System (AMPS) which is created during the execution of a LIPS program. We have derived an equivalence theorem to show that the implementation of asynchronous communication using AMPS satisfies its formal specification defined using SACS. The proof is derived by creating a weak bisimilarity relation between the LTS of SACS and AMPS. Having verified their equivalence, SACS and AMPS can be implemented in any asynchronous communicating systems with minimal modifications.

Chapter 6

Conclusion

As computer hardware is becoming cheaper, the computer users are moving away from central mainframe based computing to network based computing and distributed computing. Along with this trend, tools for developing distributed systems have also become available. This includes languages for implementing distributed systems. One such language is LIPS. It is a point-to-point asynchronous message passing programming language which is simple, secure and portable, which handles communication and computation separately. This thesis involves further development of LIPS. In this chapter, a summary of the thesis is given where the contributions are highlighted and directions for future research are presented.

6.1 Contributions to the Knowledge

The main contribution of this research is the formal semantics for LIPS using operational specifications. The outcomes of this research are the operational semantics and the abstract machine for LIPS, and Structural Operational Semantics (SOS) for the Specification of Asynchronous Communicating Systems (SACS) and Asynchronous Message Passing Systems (AMPS). Significant effort has been taken to evaluate the developed semantics and assess its accuracy and completeness. The contributions of the thesis can be summarised as follows:

1. In Chapter 3 an introduction to Asynchronous Message Passing System (AMPS) proposed by Bavan et al. [2007b] is given. The unique feature of AMPS is that it avoids the use of buffers for its asynchronous communication by using a Data Structure (DS) and a Data Matrix (DM).

   One of the main results of the work is defining the semantics and developing the code to implement AMPS into the LIPS compiler. When the LIPS compiler compiles and executes a LIPS program, a DS and a DM are created which work together to pass messages asynchronously without buffers. The compiler has been tested

for its message passing capability using simple applications. Currently AMPS has been created as a centralised system but it can be partitioned to runs on different processors. AMPS created for LIPS can be integrated with any distributed system to pass messages asynchronously.

2. In Chapter 4 operational semantics and an abstract machine are defined for LIPS. One of the distinct features of LIPS, the capability to handle computation and communication independently, has been exploited in developing the semantics and the abstract machine for LIPS.

   (a) **Operational Semantics for LIPS**

   The computation part of LIPS program is made up of 'C' programming statements (refer Table 4.3) and they can successfully be described as large steps providing direct relation between initial and final states of computation. Big-step semantics which satisfies this criteria has been used to define the operational semantics for the computation part of LIPS program.

   Communication in a LIPS program is implemented using AMPS. Big-step semantics can only specify configurations related by finite computations. On the other hand, small-step semantics or Structural Operational Semantics (SOS) contains not only the description of the initial and final states of program but also the intermediate steps of execution using labelled transitions. Because of this characteristic property, SOS has been used to define the operational behaviour of AMPS.

   By adopting this two step strategy, our approach combines the advantages of big-step and SOS.

   (b) **Abstract Machine for LIPS**

   While operational semantics is used to specify the meaning of programs, abstract machines are used to provide intermediate representation of the language's implementation. An abstract machine called LIPS Abstract Machine (LAM) has been defined using re-write rules. The LAM code has been defined in such a way that the computation and communication part of LIPS programs have been handled independently.

   (c) **Verifying the correctness of Operational Semantics with the LAM**

   The code needed for the operational semantics has been verified for its correctness against the LAM that describes the executional behaviour in this context.

LIPS compiler has been created based on the abstract machine of LIPS and tested for simple applications across platforms. The unique feature of defining the opera-

tional semantics and LAM is that both of the definitions handle the communication and computation part of LIPS programs independently. This will surely help us to

- manage the components efficiently and
- implement the AMPS in any other asynchronous communicating systems.

3. Chapter 5 considers SACS developed to specify the asynchronous message passing in LIPS which uses point-to-point communication. Though SACS [Bavan and Illingworth, 2000, Bavan et al., 2007a] has been validated, there is no formal semantics defined for SACS. This work considers defining the formal semantics mainly for two purposes. They are as follows:

   - To study the behaviour of SACS. As the behaviour of process algebra can well be described using Structural Operational Semantics (SOS), we have defined the SOS for SACS.
   - To verify the correctness of specifying the communication and its corresponding implementation using AMPS in a LIPS program.

   An equivalence theorem has been derived to show that the implementation of asynchronous communication using AMPS satisfies its specification defined using SACS. The proof uses weak bisimilarity relation between the Labelled Transition System of SACS and AMPS. Having verified their equivalence, SACS and AMPS can be implemented in any asynchronous communicating systems with minimal modifications.

6.2 Future Work

There are a number of ways in which this work can be extended and developed further. We briefly explain few of them.

1. **Denotational and Axiomatic Semantics for LIPS**

   With regard to developing the formal semantics for LIPS, this research considered only the operational semantics. While operational semantics is used to implement a language and prove the correctness of compiler implementation, denotational and axiomatic semantics are used to reason about the programs and prove properties of programs. As it is the first ever work done in defining the formal semantics for LIPS, we took the liberty of choosing operational semantics. Therefore, developing denotational and axiomatic semantics for LIPS can complement LIPS language.

2. **High level/abstract specification of LIPS**

   Specification of Asycnhronous Communicating System (SACS) used in this research

specifies only the communication part of LIPS. Although process algebra is ideal for specifying the interactions between processes, it is not particularly suitable for modelling complex data structures. SACS that was developed to model the communication part of LIPS has to be integrated with another formal technique to specify the computational part of LIPS. Integrating two specifications is not a new technique. Blending Object-Z with CCS [Taguchi and Araki, 1997]and timed CSP [Mahony and Dong, 1998] are a few examples. There are tools such as State machines, VDM, Z Object-Z, Petri nets, and Guarded Command Languages (GCL) which can be used to specify sequential programs. Computational component of a distributed programming language can very well be specified using one of these tools. Work has already been done with regard to specifying the computational part of LIPS. GCL has been chosen owing to its capability to specify using weakest pre-condition which can be used to study the axiomatic properties of LIPS. Detailed specification and integration of GCL with SACS can be found in [Rajan et al., 2006, 2007b]. As the work does not fall into the scope of this research, it has not been added included to the thesis. In continuation with this, work can be extended to identify the suitability of the combined formal tool to other distributed applications.

3. **Verifying Compiler for LIPS**

   Developing a verifying compiler for a distributed programming language like LIPS which can be used to determine the correctness of a program it compiles with respect to some specified properties, will be a major achievement. According to Hoare's concept of a verifying compiler [Hoare, 1969], source program will have the required assertions added to it at strategic points of the program for verification. The verifying compiler will prove the correctness of the program in terms of its associated assertions/specifications. So the target program will consist of some proof of correctness using which verification can be done.

   The approach we propose is different from Hoare in the sense that the programmers do not need to include assertions in the source code. LIPS program and SACS with GCL specification for the user requirements can be sent to the LIPS compiler. While generating the target code, the compiler can generate the SACS with GCL specification of the source code using some reverse engineering. This generated specification can be compared with the specifications supplied with the source code. That is, checking can be performed for the equivalence of the source and target specifications in order to verify the correctness of LIPS programs. The verification result and the target code (Java) can be generated as output. In this way, the proposed LIPS compiler can verify the correctness of the source code every time before allowing it to be executed. This approach can avoid burdening the

programmer with the task of inserting assertions into the programs. Instead, we can provide the original specification and let the compiler verify that the source program faithfully implements the specification. This idea has been published in [Rajan, 2005b, 2004, 2005a].

Appendices

Appendix A

Sample LIPS Programs

A.1 Sample LIPS program - 1: Finding the area under a curve using Simpson's rule

```
program Simpson;
begin
[1]: connect host ([result]) ->  ([Width], [Segment [0 .. 2]]);
[2]: connect Area  ([Width], [Segment[0 .. 2]]) ->   ([area[0 .. 2]]);
[3]: connect Summer  ([area [0 .. 2]]) ->  ([result]);
node host  (double result)->  (double Width, int Segment [2]) {
[ ] => { //send all area nodes the Width
        Width= .333;
        //send all area nodes their Segment number
       int i=0;
       for (i = 0; i <=2; i ++){
            Segment[i] = i+1;}
            }
[#] => {
// start the process
       }
[result] => {
        print("the result is = ", result);
        }
}
node Area  (double Width, int Segment[2]) ->  (double  area[2]){
double x, y;
[Segment[0 .. 2], Width] => {
// calculate the starting point of x
```

```
int j;
for(j=0; j<=2; j++){
x = Width * (2.0 * Segment[j] + 1.0)/2.0;
y = 4.0 / (1.0 + (x * x));
area[j] = x * y; /*outputs to channel s */
}
}
}
node Summer (double area [2]) ->  (double result) {
[area [0 .. 2]] => {
double total;
int count;
total = 0.0;
// calculate the total
for(count = 0; count <=2; count++){
total = total + area[count];}
result = total;/*send the result*/
 }
}
end.
```

A.2 Sample LIPS program - 2: Vending Machine Problem

```
program VendingMachine;
//// Define the system
////VENDING_SYSTEM = HOST x CUSTOMER x MACHINE_INTERFACE x MACHINE_INTERNALS
begin
//network definition
[1]: connect host ([]) ->  ([trayEmpty]);
[2]: connect Customer  ([deliver]) ->   ([coin], [button]);
[3]: connect Mac_Interface  ([coin], [button]) ->   ( [drkSig]);
[4]: connect Mac_Internal  ([trayEmpty], [drkSig]) ->  ([deliver]);
//node definitions
//hostnode
node host ()->  (Boolean trayEmpty) {
[#] => {
        //set machineReady to true
```

```
        trayEmpty = true;}
}
node Customer (Boolean deliver) ->   (int coin, Boolean button) {
[ ] => {
      //when machineReady he presses buttun and inserts coin
      coin=80;
      button = true;
      print(" i have pressed the button and inserted ", coin, "p");
}
[deliver]=>{
            //drink making process is going on signal is received from the
            //machine internals which will make the customer to wait
            print("drink has been delivered");
}
}
node Mac_Interface  (int coin, Boolean button) ->   (Boolean drkSig ){
[coin, button]=>{
                print("machine has received ", coin, "p");
                //set drksig
                drkSig=true;
}
}
node Mac_Internal  (Boolean trayEmpty, Boolean drkSig) ->  (Boolean deliver){
[trayEmpty, drkSig]=>{
                        if ((trayEmpty)&&(drkSig)){
                            print ("making drink.............");
                            settimer(20);
                            deliver=true;
                        }
                        print("DRINK READY");
}
}
end.
```

Appendix B

Case Study - 2 - Post Office Scenario

Consider a post office which has many counters open. The post office deals with three items which it provides to its customers. There is a queue of customers. The length of the queue at any given time is from 0 to n. The assumption is that the postoffice has an unlimited stock of items to sell. The stockroom automatically replenishes items for each counter as they run out. In order to model this system, it is assumed that the post office has a queue/counter controller to oversee that counters are not idle when the queue is not empty. The processes involved in the scenario are explained below:

Customer: When a customer comes into the post office, he/she checks whether any one of the counters is vacant.

Counter: When a counter finishes servicing a customer, a message is sent to the controller that displays the status of the counter. When a server at a counter gets new customer, a signal is sent to the controller notifying that the counter is busy. When the items are sold, the counter sends a signal to the stock room about the number of items sold. If the quantity of an item becomes less than 5 then 20 of those items will be transferred from the stock room to the counter. If there are no customers in the queue then nothing happens.

Controller: The controller is responsible for displaying the status of each counter. When it receives signals from a counter informing that it has completed its service with a customer, the controller displays a 'vacant' message for the corresponding counter. In the similar manner, when the controller receives a signal from a counter servicing a new customer, it displays a 'busy' message against that counter. This also means that, if there is more than one free counter, a customer can choose any one of the free counters.

Stockroom: The stockroom has unlimited stock of all three items sold. When it receives signals from the counters indicating the quantities of items sold, the stockroom replenishes the stock levels of the respective counters. A pictorial representation of one possible solution is shown in Figure B.1. As explained in the case study for vending machine in Section 3.5.1, when the LIPS program for the post office problem is compiled, the compiler

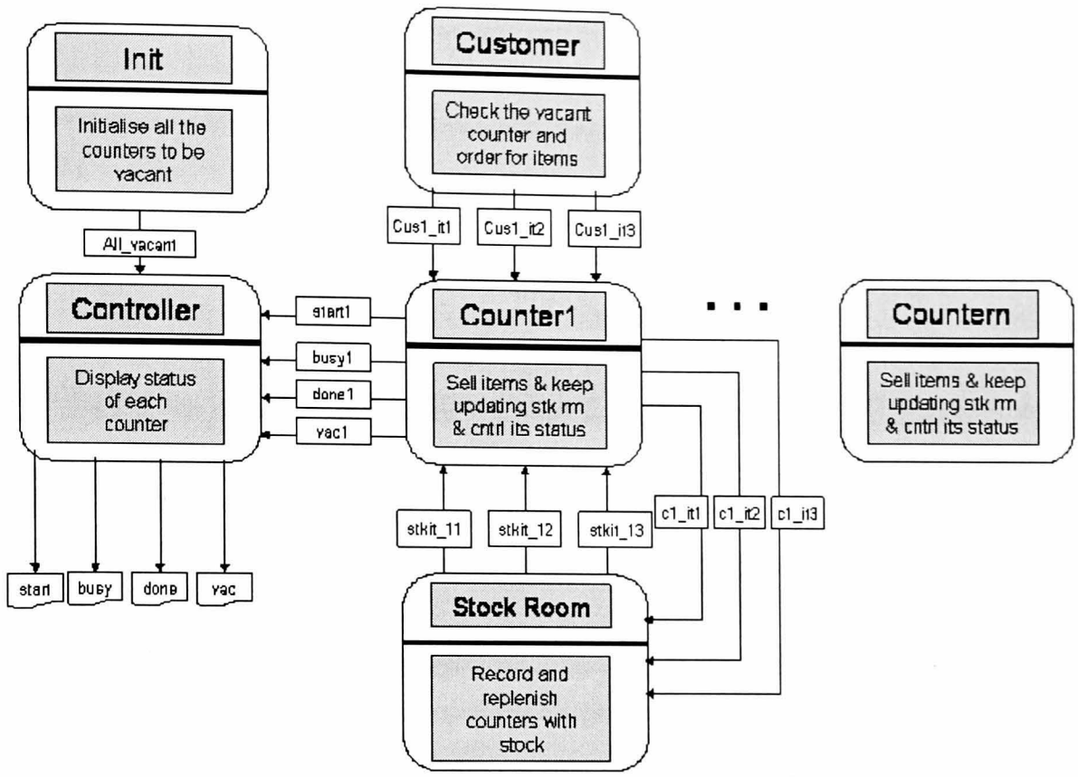


Figure B.1: Pictorial representation for the Post Office Problem.

generates the Driver Matrix(DM) and the Data structure (DS). The DS is initialised with `null` values. The DM is shown in Table B.1 and it's corresponding DS is shown in Figure B.2.

Table B.1: Driver Matrix for the Post Office Problem

vnum	SrcNodeNum	type	Destination nodes				
0	1	4	0	1	0	0	0
1	2	4	0	0	1	0	0
2	3	1	0	0	0	1	0
3	3	1	0	0	0	1	0
4	3	1	0	0	0	1	0
5	4	4	0	1	0	0	0
6	4	1	0	0	0	0	1
7	4	1	0	0	0	0	1
8	4	1	0	0	0	0	1
9	4	1	0	0	1	0	0
10	4	1	0	0	1	0	0
11	4	1	0	0	1	0	0
12	4	4	0	1	0	0	0
13	5	1	0	0	0	1	0
14	5	1	0	0	0	1	0
15	5	1	0	0	0	1	0

```
        Data Structure for the Post Office Problem
        1       Host
input list finished
0       All_Vacant      0       null
output list finished
        2       CONTROLLER
0       All Vacant      0       null
5       start   0       null
12      done    0       null
input list finished
1       vac_busy1       0       null
output list finished
        3       CUSTOMER
1       vac busy1       0       null
9       cus1 rit1       0       null
10      cus1 rit2       0       null
11      cus1 rit3       0       null
input list finished
2       cus1 it1        0       null
3       cus1 it2        0       null
4       cus1 it3        0       null
output list finished
        4       COUNTER1
2       cus1 it1        0       null
3       cus1 it2        0       null
4       cus1 it3        0       null
13      stkit 11        0       null
14      stkit 12        0       null
15      stkit 13        0       null
input list finished
5       start   0       null
6       c1 it1  0       null
7       c1 it2  0       null
8       c1 it3  0       null
9       cus1 rit1       0       null
10      cus1 rit2       0       null
11      cus1_rit3       0       null
12      done    0       null
output list finished
        5       STOCKROOM
6       c1 it1  0       null
7       c1 it2  0       null
8       c1_it3  0       null
input list finished
13      stkit 11        0       null
14      stkit 12        0       null
15      stkit 13        0       null
output list finished
```

Figure B.2: Data Structure of the AMPS.

Bibliography

Foundational Calculi for Programming Languages, 1995.

W. B. Acherman, J. B. Dennis, and William B Ackerman. Val- oriented algorithmic language, preliminary reference manual. Technical report, Cambridge, MA, USA, 1979.

S. Ahuja, N. Carriero, and N. Gelernter. Linda and friends. *IEEE computer*, 19(8):26–34, 1986.

Elvira Albert, Michael Hanus, Frank Huch, Javier Olvier, and Germán Vidal. An operational semantics for declarative multi-paradigm languages. In *Proc. of the Int'l Workshop on Reduction Strategies in Rewriting and Programming (WRS 2002)*, volume 70 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2002.

D. G. Andrew and M. P. Andrew. *Higher Order Operational Techniques in Semantics*. Publications of the Newton Institute, Cambridge University Press, 1998. ISBN 0521631688.

G. R. Andrews and F. B. Schneider. Concepts and notations for concurrent programming. *ACM Comput. Surv.*, 15(1):3–43, 1983.

Gregory R. Andrews. Distributed programming languages. In *ACM 82: Proceedings of the ACM '82 conference*, pages 113–117, New York, NY, USA, 1982. ACM. ISBN 0-89791-085-0. doi: http://doi.acm.org/10.1145/800174.809772.

Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, July 2007. ISBN 193435600X. URL `http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20\&path=ASIN/193435600X`.

I. Attali, D. Caromel, S. O. Ehmety, and S. Lippi. Semantic based visualisation for parallel object-oriented programming. *OOPSLA'96, ACM Press, Sigplan Notices, San Jose, CA*, 18(6):711–729, 1996.

H. E. Bal. Orca: a portable user-level shared object system. Technical report, Technical Report IR-408, Dept. of Mathematics and Computer Science, Vrije Universiteit, 1996.

H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, 1989.

Henri E. Bal. *Programming distributed systems*. Silicon Press, Summit, NJ, USA, 1990. ISBN 0-929306-05-8.

A. S. Bavan and E. Illingworth. Design and implementation of reliable point-to-point asynchronous message passing system. In *Proceedings of The 10th International Conference on Computing and Information ICCI '2000, Kuwait, (18th–21st November 2000).* ICCI, 2000.

A. S. Bavan and E. Illingworth. *A Language for Implementing Parallel and distributed Systems using asynchronous point-to-point communication.* Nova Science Publishers, Inc., Commack, NY, USA, 2001. ISBN 1-59033-116-8.

A. S. Bavan, E. Illingworth, A. V. S. Rajan, and G. Abeysinghe. Specification of asynchronous communicating systems (sacs). In *Proceedings of IADIS International Conference Applied Computing 2007, Salamanca, Spain (18th–20th February 2007).* IADIS, 2007a. URL `http://www.pms.ifi.lmu.de/publikationen/#REWERSE-RP-2007-012`.

A. S. Bavan, A. V. S. Rajan, and G. Abeysinghe. Asynchronous message passing architecture for a distributed programming language. In *Proceedings of IADIS International Conference Applied Computing 2007, Salamanca, Spain (18th–20th February 2007).* IADIS, 2007b. URL `http://www.pms.ifi.lmu.de/publikationen/#REWERSE-RP-2007-012`.

D. Berry, R. Milner, and D. N. Turner. A semantics for ml concurrency primitives. In *Annual Symposium on Principles of Programming Languages, Proceedings of the 19th ACM SIGPLAN-SIGACT*, pages 119–129, 1992.

E. Best, R. Devillers, and M. Koutny. *Petri Nets, Process Algebras and Concurrent Programming Languages*, volume 1492 of *Lectures on Petri Nets II: Applications*, pages 1–84. Springer verlag, 1998.

D. Bjorner and CB Jones. The vienna development method: The meta-language. *LNCS Springer Verlag*, 61, 1978.

W. Brauer, D. B. Hansen, D. Gries, C. Moler, G. Seegmueller, J. Stoer, and N. Wirth. The programming language Ada, reference manual. *Lecture Notes in Computer Science, Springer-Verlag*, 106, 1981.

P. Brinch-Hansen. The programming language concurrent pascal. *IEEE Transactions on Software Engineering*, SE-1(2):199–207, 1975.

A. Burns, A. M. Lister, and A. J. Wellings. *A Review of Ada Tasking.* Springer-Verlag, New York, Inc., 1987. ISBN 3-540-18008-7.

A. Burns, A. J. Wellings, A. M. Koelmans, M. Koutny, A. Romanovsky, and A. Yakovlev. On developing and verifying design abstractions for reliable concurrent programming in Ada. volume XXI, pages 48–55, New York, NY, USA, 2001. ACM. doi: http://doi.acm.org/10.1145/374369.374381.

R. Calkin, R. Hempel, H. C. Hoppe, and P. Wypior. Portable programming with parmacs message passing library. *Parallel Computing*, 20(4):615–632, 1994.

Rachel Cardell-Oliver. An equivalence theorem for the operational and temporal semantics of real-time, concurrent programs. *J. Log. Comput.*, 8(4):545–567, 1998.

Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *FoSSaCS '98: Proceedings of the First International Conference on Foundations of Software Science and Computation Structure*, pages 140–155, London, UK, 1998. Springer-Verlag. ISBN 3540643001. URL `http://portal.acm.org/citation.cfm?id=759638`.

W. E. Carlson, L. E. Druffel, D. A. Fisher, and W. A. Whitaker. Introducing ada. In *ACM 1980 Annual Conference ACM '80*, pages 263–271, New York, , NY, 1980. ACM Press.

N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4): 444–458, 1989.

N. Carriero, D. Gelernter, and J. Leichter. Distributed data structures in linda. In *POPL '86: Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 236–242, St. Petersburg Beach, Florida, 1986. ACM Press.

M. H. Cheheyl, M. Gasser, G. A. Huff, and J. K. Millen. Verifying security. *ACM Computing Surveys*, 13(3), 1981.

Liang Chen, Stuart Anderson, and Faron Moller. A timed calculus of communicating systems. Technical Report ECS-LFCS-90-127, Department of Computer Science, University of Edinburgh, Department of Computer Science University of Edinburgh The King's Buildings Edinburgh EH9 3JZ, 1990. URL `http://www.lfcs.inf.ed.ac.uk/reports/90/ECS-LFCS-90-127/ECS-LFCS-90-127.pdf`.

Rance Cleaveland and Scott A. Smolka. Priorities in process algebra. *Information and Computation*, 87:58–77, 1990.

R. P. Cook. *mod - a language for distributed programming. *IEEE Transactions on Software Engineering*, 6(6):563–571, 1980.

R. L. Crole. Operational semantics, abstract machines and correctness. Technical report, Lecture Notes for the Midlands Graduate School in the Foundations of Computer Science, 2006.

D. E. Culler, A. Dusseau, S. Goldstein C., A. Krishnamurthy, S. Lumetta, T. Eicken, and K. Yelick. Parallel programming in split c. 1993.

O. Danvy. A rational deconstruction of landin's secd machine. Technical Report RS-03-33, BRICS Report Series Publications, 2003.

E. Demaine. First class communication in mpi. In *Proceedings of the Second MPI Developers COnference (MPIDC'96)*, pages 189–194, Los Alamitos, CA, USA, 1996. IEEE Computer Society. ISBN 0-8186-7533-0.

E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

K. Elsom. Asynchronous communication in Ada. In *Proceedings of the third international workshop on Real-time Ada issues*, pages 57–65, Farmington, Pennsylvannia, United States, 1989. ACM Press.

Wolfgang Ertel. Performance of competitive or-parallelism. In *ICLP '91: Pre-Conference Workshop on Parallel Execution of Logic Programs*, pages 132–145, London, UK, 1991. Springer-Verlag. ISBN 3-540-55038-0.

C. Fencott. *Formal Methods for Concurrency.* International Thompson Computer Press, 1996.

M. Fernandez. *Programming Languages and Operational Semantics-An Introduction.* King's College Publications, London,UK, 2004. ISBN 0954300637.

Gianluigi Ferrari, Roberto Guanciale, and Daniele Strollo. JSCL: A Middleware for Service Coordination. In *Proceedings of FORTE 2006, 26th IFIP WG 6.1 International Conference on Formal Methods for Networked and Distributed Systems*, volume 4229 of *Lecture Notes in Computer Science*. Springer Verlag, 2006. URL `http://www.di.unipi.it/~giangi/forte06.pdf`.

C. J. Fidge. A formal definition of priority in csp. *ACM Trans. Program. Lang. Syst.*, 15 (4):681–705, 1993.

R. W. Floyd. Nondeterministic algorithms. *Journal of the ACM*, 14:636–644, 1967.

I. Foster and K. M. Chandy. Fortran m: A language for modular parallel programming. *Parallel and Distributed Computing*, 26(1):24–35, 1995.

Cedric Fournet and Georges Gonthier. The join calculus: A language for distributed mobile programming. In *In Proceedings of the Applied Semantics Summer School (APPSEM), Caminha*, pages 268–332. Springer-Verlag, 2000.

Cedric Fournet, Fabrice Le Fessant, Luc Maranget, and Alan Schmitt. Jocaml: a language for concurrent distributed and mobile programming. In *Summer School on Advanced Functional Programming, Lecture Notes in Comput. Sci. 2638*, pages 129–158. Springer-Verlag, 2002.

Michael Friendly. *Advanced Logo, A Language for Learning.* Lawrence Erlbaum Associates, 1 edition (13 jul 1988) edition, 1988.

V.C. Galpin. *Equivalence semantics for concurrency: comparison and application.* PhD thesis, ECS-LFCS-98-397, Department of Computer Science, University of Edinburgh, 1998.

N. H. Gehani. Message passing in concurrent c: Synchronous versus synchronous. *Journal of Software Practice and Experience*, 20(6):571–592, 1990.

N. H. Gehani and W. D. Roome. Implementing concurrent c. *Software - Practice and Experience*, 22(3):265–285, 1992.

C. M. Geschke, J. H. Morris Jr., and E. H. Satterthwaite. Early experience with mesa. *Communications of the ACM,*, 20(8):540–553,, 1977.

S. Glesner. Asms versus natural semantics:a comparison with new insights. In E. Riccobene E. Boerger, A. Gargantini, editor, *Abstract State Machines 2003. Advances in Theory and Practice: 10th International Workshop, ASM 2003*, volume 2589, pages 293–309, Taormina, Italy, 2003. Springer Berlin / Heidelberg.

G. Goldszmidt, S. Katz, and S. Yemini. Interactive blackbox debugging for concurrent languages. In *In Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging (Madison, Wisconsin, United States, May 05 - 06, 1988). R. L. Wexelbalt, Ed. PADD '88*, pages 271–282, New York, 1988. ACM Press.

A. D. Gordon. *Operational equivalences for untyped and polymorphic object calculi*, pages 9–54. Higher order operational techniques in semantics. Cambridge University Press, 1998. ISBN 0-521-63168-8.

D. Gray. *Introduction to the Formal Design of Real-Time Systems.* Springer, Paperback, 2000. ISBN 3540761403.

J. V. Guttag, J. J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specification.* 1993.

J. Hannan and D. Miller. From operational semantics to abstract machines. *Mathematical Structures*, 2(4):415–459, 1992.

B. P. Hansen. Distributed processes: A concurrent programming concept. In *Comm. ACM*, volume 21, pages 934–941, New York, NY, USA, 1978. ACM Press.

Seif Haridi, Peter Van Roy, Per Brand, and Christian Schulte. Programming languages for distributed applications. *New Generation Computing*, 16(3):223–261, 1998. URL `citeseer.ist.psu.edu/article/haridi98programming.html`.

Wilhelm Hasselbring. Programming languages and systems for prototyping concurrent applications. *ACM Comput. Surv.*, 32(1):43–79, 2000. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/349194.349199.

Douglas Heintzman. An introduction to open computing, open standards, and open source, July 2003. URL `http://www.ibm.com/developerworks/rational/library/1303.html`.

C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580 and 583, 1969.

C. A. R. Hoare. Communicating sequential processes. *Communications of the Association of Computing Machinery*, 21(8):666–677, 1978.

S. Holmstrom. Pfl: A functional language for parallel programming. *Declarative Programming Workshop, Programming Methodology Group, Chalmers University of Technology, University of Goteborg, Sweden*, 1983.

Waldemar Horwat. A concurrent smalltalk compiler for the message-driven processor. Technical report, Cambridge, MA, USA, 1988.

N. C. Hutchinson. Emerald: An object-based language for distributed programming. Technical Report TR 87-01-01,, PhD thesis, 1987.

G. Hutton and J. Wright. Calculating an exceptional machine. In *Proceedings of the Fifth Symposium on Trends in Functional Programming*, Munich, Germany, 2005.

D. C. Hyde. Introduction to the programming langauge occam, 1995.

E. Illingworth, A. S. Bavan, and G. S. Flora. An asynchronous harness for transputer systems that does not use polling techniques. In P. Fritzson and L. Finmo, editors, *Parallel Programming and Applications*, pages 364–369. IOS Press, 1995.

Inmos. *Occam 2.1 Reference Manual.* Prentice-Hall, England, 1988.

J. Jacky. *THE WAY OF Z : Practical programming with formal methods.* Cambridge University Press, 1997. ISBN 0-521-5597.

H. Jansohn. Ada for distributed systems. *AdaLett*, VIII(7):101–103, 1988.

A. Jeffrey. *Semantics for core Concurrent ML using computation types*, pages 55–90. Higher order operational techniques in semantics. Cambridge University Press, 1998. ISBN 0-521-63168-8.

Alan Jeffrey. A fully abstract semantics for a concurrent functional language with monadic types, 1995.

C. B. Jones. The transition from vdl to vdm. *Journal of Universal Computer Science*, 7 (8):631–640, 2001.

Simon P. Jones. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, May 2003. ISBN 0521826144. URL `http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20\&path=ASIN/0521826144`.

S. Kamran. Concurrent c/c++ programming langauges, 1996.

A. A. C. Klaiber and A. H. M. Levy. A comparison of message passing and shared memory architectures for data parallel programs. In *Proceedings of the 21ST annual international symposium on Computer architecture, Chicago, Illinois, United States*, pages 94–105, Chicago, Illinois, United States, 1994. IEEE Computer Society Press.

Bartosz Klin. An abstract coalgebraic approach to process equivalence for well-behaved operational semantics, 2004.

J. Kramer. Distributed software engineering. pages 253–263, Sorrento, Italy, 1994. IEEE Computer Society Press.

D. Krizanc and A. Saarimaki. Bulk synchronous parallel: Practical experience with a model for parallel computing. In *PACT '96: Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques*, page 208, Washington, DC, USA, 1996. IEEE Computer Society.

J. D. Kubiatowicz. Integrated shared-memory and message-passing communication in the alewife multiprocessor, 1998. URL `http://www.cs.berkeley.edu/~kubitron/papers/alewife/pdf/kubi-phdthesis.pdf`.

P. Landin. An abstract machine for designers of computing languages. *IFIP Congress*, pages 438–439, 1965.

P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1963.

H. Ledgard. *Reference Manual for the ADA Programming Language.* Springer-Verlag New York, Inc., New York,, 1983. ISBN 0387908870.

C. Lin and L. Snyder. A comparision of programming models for shared memory multiprocessors. *In proceedings of the 1990 International Conference on Parallel Processing,* (II):163–170, 1990.

L. Logrippo, T. Melanchuck, and R. J. D. Wors. An algebraic specification language LOTOS: An industrial experience. In *Proc. ACM SIGSOFT Int'l. Workshop on Formal Methods in Software Development,* pages 59–66, 1990.

F. J. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Equivalence of two formal semantics for functional logic programs. *Electron. Notes Theor. Comput. Sci.*, 188:117–142, 2007. ISSN 1571-0661. doi: http://dx.doi.org/10.1016/j.entcs.2007.05.042.

J. Magee, N. Dulay, and J. Kramer. Regis: A constructive development environment for parallel and distributed programs. *Distributed Systems Engineering Journal, Special Issue on Configurable Distributed Systems,* 1(5):304–312, 1994.

J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In W. Schafer and P. Botella, editors, *Proc. 5th European Software Engineering Conf.*, volume 989, pages 137–153, Sitges, Spain, 1995. Springer-Verlag, Berlin.

Jeff Magee, Naranker Dulay, and Jeff Kramer. Structuring parallel and distributed programs. *Software Engineering Journal,* 8:73–82, 1993.

B. Mahony and J. S. Dong. Blending object-z and timed csp: An introduction to tcoz. *The 20th International Conference on Software Engineering,* 1998.

Slawomir P. Maludzinski and Grzegorz Dobrowolski. Agent environment and knowledge in distributed join calculus. In *CEEMAS '07: Proceedings of the 5th international Central and Eastern European conference on Multi-Agent Systems and Applications V,* pages 298–300, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-75253-0. doi: http://dx.doi.org/10.1007/978-3-540-75254-7_30.

J. McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer programming and Formal Systems,* pages 21–28, North-Holland, 1963.

J. McCarthy and J. A. Painter. Correctness of a compiler for arithmetic expressions, mathematical aspects of computer science. In J. T. Schwartz, editor, *Porc. Symp. in Applied Mathematics,* volume 19, pages 33–41. Providence, RI: American Mathematical Society, 1967.

John McCarthy. Towards a mathematical science of computation. In *IFIP Congress,* pages 21–28, 1962.

R. Milner. *A Calculus of Communicating Systems.* Springer Verlag, New York Inc.,, 1982. ISBN 0387102353.

R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML.* MIT Press, 1 edition, 1990.

R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML - Revised.* MIT Press, 1997.

Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus.* Cambridge University Press, June 1999. ISBN 0521658691. URL `http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20\&path=ASIN/0521658691`.

Robin Milner. The polyadic ?-calculus: a tutorial. Technical report, LOGIC AND ALGEBRA OF SPECIFICATION, 1991.

K. Mitchell. Concurrency in a natural semantics. Technical report, LFCS report ECS-LFCS-94-311, 1994.

G. Morrisett and R. Harper. *Semantics of memory management for polymorphic languages*, pages 175–226. Higher order operational techniques in semantics. Cambridge University Press, 1998. ISBN 0-521-63168-8.

P. D. Mosses. Exploiting labels in structural operational semantics. Technical report, BRICS publications, IOS press, 2005.

T. Ngo and L. Snyder. On the influence of programming models on shared memory computer performance. *In scalable High Performance Computing Conference*, pages 284–291, 1992.

N. S. Papaspyrou. *A Formal Semantics for the C Programming Language.* PhD thesis, National Technical University of Athens, 1998.

David Park. Concurrency and automata on infinite sequences. *Lecture Notes in Computer Science, Springer-Verlag*, page 167, 1981.

J. C. Peralta, J. P. Gallagher, and H. Saglam. Analysis of imperative programs through analysis of constraint logic programs. In G. Levi, editor, *Static Analysis, 5th International Symposium, SAS'98*, pages 246–261, Pisa, Italy, 1998. Springer Verlag.

M. (Ed). Pettersson. *Compiling Natural Semantics*, volume Lecture Notes in Computer Science. Springer Verlag, 1999. ISBN 3-540-65968-4.

Keshav Pingali and Kattamuri Ekanadham. Accumulators: A new logic variable abstractions for functional languages. In *FSTTCS*, pages 377–399, 1988.

G. Plotkin. A structured approach to operational semantics. Technical report, Technical Report DAIMI FN-19, 1981.

G. Plotkin. The origins of structural operational semantics, 2003. URL `citeseer.ist.psu.edu/plotkin03origins.html`.

Sanjiva Prasad and S. Arun-Kumar. Introduction to operational semantics. In *The Compiler Design Handbook*, pages 841–890. 2002.

A. V. S. Rajan, S. Bavan, and G. Abeysinghe. Semantics for a distributed programming language using sacs and weakest pre-conditions. In *ADCOM 2006, the 14th International Conference on Advanced Computing and Communication*, Mangalore, India, 2006. IEEE press.

A. V. S. Rajan, A. S. Bavan, and G. Abeysinghe. *Semantics for an Asynchronous Message Passing System*, volume XVIII of *Advances and Innovations in Systems, Computing Sciences and Software Engineering*. Springer, 2007a. ISBN 978-1-4020-6263-6.

A. V. S. Rajan, S. Bavan, and G. Abeysinghe. Semantics for a distributed programming language using sacs and weakest pre-conditions. *International Journal of Information Processing(IJIP)*, 1(1):I.K. Published Journals, 2007b.

A.V.S. Rajan. Verifying compiler generation for a domain specific language. In *In proceedings of International Conference on Functional Programming*. ACM Press, 2004.

A.V.S. Rajan. Software verification for parallel/distributed systems. In *Proceedings of PREP 2005*. The Engineering and Physical Sciences Research Council (EPSRC), 2005a.

A.V.S. Rajan. Developing a verifying compiler for lips. In *In Proceedings of SIGCSE*. ACM Press, 2005b. ISBN 1-58113-997-7.

Linda Rising. Tasking troubles and tips (abstract). In *CSC '88: Proceedings of the 1988 ACM sixteenth annual conference on Computer science*, pages 729–730, New York, NY, USA, 1988. ACM. ISBN 0-89791-260-8. doi: http://doi.acm.org/10.1145/322609.323187.

D. A. Schmidt. *Denotational Semantics: A Methodology for Language Developement*. Allyn & Bacon, Inc.,, Boston, 1986.

Dana Scott and Christopher Strachey. Toward a mathematical semantics for computer languages. Programming Research Group Technical Monograph PRG-6, Oxford Univ. Computing Lab., 1971.

N. B. Serbedzija. Asynchronous communication in occam. *ACM SIGPLAN Notices*, 23 (12):51–62, 1988.

E. Y. Shapiro. Concurrent prolog: A progress report. *IEEE Computer*, 19(8):44–58, 1986.

D. B. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, 1998.

K. Slonneger and B. L. Kurtz. *Formal Syntax and Semantics of Programming Languages-a Laboratory-based Approach*. Addison Wesley, 1995.

P. Stenstram and F. Dahlgren. Applications for shared memory multiprocessors. *Computer*, 29(12):29–31, 1996.

M. Strecker. Formal verification of a java compiler in isabelle. In A. Voronkov, editor, *18th International Conference on Automated Deduction*, pages 63–77, Copenhagen, Denmark, 2002. LNAI 2382, Springer.

R. Strom and S. Yemini. Nil: an integrated language and system for distributed programming. pages 73–82, 1983.

R. Strom and S. Yemini. The nil distributed systems programming language: A status report. *ACM Sigplan Notices*, 20(5):36–43, 1985.

K. Taguchi and K. Araki. The state-based ccs semantics for concurrent z specification. In *1st Internation Conference on Formal Engineering Methods (ICFEM'97)*, 1997. URL `http://icaps03.itc.it/tutorials/tutorial4.htm,Accessedon22/07/2004`.

A. S. Tanenbaum, M. F. Kaashoek, K. G. Langendoen, and C. J. H. Jacobs. The design of very fast portable compilers. *ACM SIGPLAN Not.*, 24(11):125 – 131, 1989.

G. Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, Cambridge. 2000.

G. K. Theodoropoulos, G. K. Tsakogiannis, and J. V. Woods. Occam: An asynchronous hardware description language? pages 249–256, 1997.

E. Tuosto. *Non Functional Aspects of Wide area Network Programming.* PhD thesis, Dipartimento di Informatica, Univ. Pisa, 2003.

David von Oheimb. Axiomatic semantics for Java^{light} in Isabelle/HOL. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs.* Technical Report 269, 5/2000, Fernuniversität Hagen, Fernuniversität Hagen, 2000. ECOOP2000 Workshop proceedings available from `http://www.informatik.fernuni-hagen.de/pi5/publications.html`.

P.H. Welch and F.R.M. Barnes. Mobile Barriers for occam-pi: Semantics, Implementation and Application. In J.F. Broenink, H.W. Roebbers, J.P.E. Sunter, P.H. Welch, and D.C. Wood, editors, *Communicating Process Architectures 2005*, volume 63 of *Concurrent Systems Engineering Series*, pages 289–316, Amsterdam, The Netherlands, September 2005. IOS Press. ISBN: 1-58603-561-4.

A. Wijngaarden. Revised report of the algorithmic language algol 68. Technical Report Sup 47, Mountain View, CA, United States, 1981.

Glynn Winskel. *The formal semantics of programming languages: an introduction.* MIT Press, Cambridge, MA, USA, 1993. ISBN 0-262-23169-7.